MANNING

Broadview®
www.broadview.com.cn

深入 RabbitMQ

[美] Gavin M.Roy 著
汪佳南 郑天民 译

RabbitMQ In Depth

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

深入RabbitMQ

RabbitMQ In Depth

[美] Gavin M.Roy 著
汪佳南 郑天民 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书对 RabbitMQ 这一业界主流的消息中间件做了全面介绍，给出了如何使用 RabbitMQ 构建消息通信系统的方法和实践。本书从 AMQP 协议出发，深入介绍各种消息属性，给出 RabbitMQ 在发送和消费消息上的特性和最佳实践，并阐述基于 RabbitMQ 所特有的交换器组件实现灵活的消息路由机制。同时，本书也讨论了如何利用 RabbitMQ 强大的集群机制实现分布式环境下的消息通信，并展示了如何在 RabbitMQ 中，使用其他传输协议以及数据库集成等功能来实现各种定制化需求。

本书的读者对象为从事互联网行业中各种分布式和服务化系统开发的研究人员、高等院校计算机相关专业的研究生和本科生，以及广大的 IT 爱好者。

版权贸易合同登记号　图字：01-2017-8361

图书在版编目（CIP）数据

深入 RabbitMQ /（美）加文·罗伊 (Gavin M. Roy) 著；汪佳南，郑天民译 .
— 北京：电子工业出版社，2018.6
书名原文：RabbitMQ in Depth
ISBN 978-7-121-34180-9

Ⅰ. ①深… Ⅱ. ①加… ②汪… ③郑… Ⅲ. ① JAVA 语言－程序设计 Ⅳ. ① TP312.8

中国版本图书馆 CIP 数据核字（2018）第 099199 号

策划编辑：张春雨
责任编辑：牛　勇
特约编辑：顾慧芳
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编　100036
开　　本：787×980　1/16　印张：15.75　字数：302.4千字
版　　次：2018 年 6 月第 1 版
印　　次：2018 年 6 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 fag@phei.com.cn。

译者序

当下互联网行业中的各种分布式和服务化系统的开发本质上就是解决一个问题，即系统如何进行拆分和集成。服务拆分需要用到面向领域思想，而服务集成则可以采用 RPC、REST、消息通信等多种技术体系。从软件设计角度讲，无论是 RPC 还是 REST 都存在一定的耦合度问题。耦合度包括技术耦合、空间耦合和时间耦合等不同的表现形式，而消息通信机制能够降低这几种耦合度。

消息通信机制在消息发送方和消息接收方之间添加了存储转发（Store and Forward）功能。通过存储转发功能，消息发送方和消息接收方之间并不需要知晓对方的存在，也不需要同时在线，更不会限制必须采用同样的实现技术。紧耦合的单阶段远程方法调用因而转变成松耦合的两阶段过程，技术、空间和时间上的约束凭借中间层得到显著缓解。

然而消息通信机制并没有想象的那么简单。我们需要在消息的生产者和消费者之间建立有效的通信链路并确定双方的通信协议。消息的消费者通常采取主动拉取和被动消费这两种模式实现消息消费；同时，出于稳定性的考虑，消费者还需要提供限流的能力。消息发送方的逻辑则相对简单，一旦消息发送出去之后，它将依赖路由规则，最终投递给符合条件的一个或多个消费者。这些构成了消息通信机制所应该具备的核心组件。

围绕消息通信机制的这些核心组件，业界存在一批关于消息通信的设计规范。而基于这些规范，不同厂商也提供了多种消息通信系统实现方案。本书所介绍的是基于高级消息队列协议（Advanced Message Queuing Protocol，AMQP）规范的 RabbitMQ，在内容上详细阐述了 RabbitMQ 的以下几个主题：

- AMQP 规范以及消息定义。
- 消息发送的过程以及可靠消息投递机制。
- 消息消费的过程以及消费者性能优化方法。
- 交换器组件以及消息路由机制。

- RabbitMQ 分布式集群构建。
- RabbitMQ 在系统集成上的具体应用。

目前，RabbitMQ 在各大互联网公司中应用十分广泛。通过 RabbitMQ 所提供的丰富的交互 API、友好的管理界面以及与生俱来的分布式特性，我们可以轻松构建一个强大的消息通信系统。然而消息通信系统的构建一方面降低了耦合性，另一方面也不可避免地引入了复杂性。如果使用不当，反而会引发各种问题。本书深入分析消息通信的各个方面，不仅介绍了 RabbitMQ 的各项基本功能，更为重要的是提供了一系列面向实战的最佳实践，可以作为广大技术人员的开发指南。

在整个翻译过程中，我们首先要感谢张春雨和顾慧芳编辑的辛苦工作，是你们的细心指导才让本书得以最终呈现给读者。其次要感谢我们的家人和朋友，没有你们的体谅和关怀，我们无法专心致志工作。最后要感谢这段经历，本书是我们首次尝试以合译的方式来完成的，当中充满了乐趣和挑战。

本书第 1 章至第 5 章是由郑天民负责翻译的，第 6 章至第 10 章是由汪佳南负责翻译的。由于时间仓促，译者水平和经验有限，书中难免有欠妥和错误之处，恳请读者批评指正。

郑天民、汪佳南

2018 年 3 月于杭州

关于作者

Gavin M. Roy 是一位积极的开源传播者和倡导者，自 20 世纪 90 年代中期就一直活跃在互联网和企业级技术之中。

关于封面

本书封面上的图片为“来自克罗地亚斯里耶姆的 Mikanovac 的男士”。该插图摘自 19 世纪中期克罗地亚传统服饰专辑。这幅出自 Nikola Arsenovic 之手的作品于 2003 年在克罗地亚斯普利特的民族博物馆展出。这些插图是从斯普利特民族博物馆的一位乐于助人的图书管理员手中获得的。它位于镇中心的中世纪罗马核心：大约公元 304 年左右罗马皇帝退休后宫殿的废墟，并伴有服饰和日常生活的描述。

着装规范和生活方式在过去的 200 年里历经变化。当时如此丰富多彩的地区多样性已不复存在。现在很难区分不同大陆的居民，更不用说不同的小村庄或仅隔几英里的小镇了。也许我们牺牲了文化多样性换来了更加多样化的个人生活。确切地说，换来的是更多样化和快节奏的科技生活。Manning 将两百年前区域生活的丰富多样性作为书籍封面来庆祝计算机业务的发明和创造，而来自这类古老书籍和藏品的插图让封面重焕生机。

目 录

第一篇 RabbitMQ和应用程序体系结构

第1章 RabbitMQ基础 3

1.1 RabbitMQ特性以及好处 4

1.1.1 RabbitMQ与Erlang 5

1.1.2 RabbitMQ与AMQP 6

1.2 谁在使用RabbitMQ，在怎么用 7

1.3 松耦合架构的优势 8

1.3.1 解耦你的应用 10

1.3.2 解耦数据库写入 11

1.3.3 无缝添加新功能 12

1.3.4 复制数据与事件 12

1.3.5 多主(Multi-Master)互联化数据与事件 13

1.3.6 高级消息队列模型 14

1.4 小结 16

第2章 使用AMQ协议与Rabbit进行交互 18

2.1 AMQP作为一种RPC传输机制 19

2.1.1 启动会话 20

2.1.2 调整正确的信道 20

2.2 AMQP RPC帧结构 21

2.2.1 AMQP帧组件 21

2.2.2 帧类型 22
2.2.3 将消息编组成帧 23
2.2.4 方法帧结构 24
2.2.5 内容头帧 26
2.2.6 消息体帧 26
2.3 使用协议 27
2.3.1 声明交换器 27
2.3.2 声明队列 28
2.3.3 绑定队列到交换器 29
2.3.4 发布消息到RabbitMQ 29
2.3.5 从RabbitMQ中消费消息 30
2.4 用Python编写消息发布者 32
2.5 从RabbitMQ中获取消息 36
2.6 小结 37

第 3 章 消息属性详解 38

3.1 合理使用属性 39
3.2 使用content-type属性创建显式的消息契约 41
3.3 通过gzip和content-encoding属性压缩消息大小 43
3.4 使用message-id和correlation-id引用消息 45
3.4.1 Message-id 45
3.4.2 Correlation-id 45
3.5 创建时间:timestamp属性 46
3.6 消息自动过期 47
3.7 使用delivery-mode平衡速度和安全性 48
3.8 使用app-id和user-id验证消息来源 49
3.8.1 app-id 50
3.8.2 user-id 51
3.9 使用type属性获取明细 51
3.10 使用reply-to属性实现动态工作流 52
3.11 使用消息头自定义属性 53
3.12 优先级属性 54

3.13 不能使用的属性:cluster-id/reserved 54
3.14 小结 55

第4章 消息发布的性能权衡 58

4.1 平衡投递速度与可靠投递 59
4.1.1 如果没有保证机制我们能期待什么 60
4.1.2 使用mandatory设置,RabbitMQ将不接受不可路由消息 62
4.1.3 发布者确认作为事务的轻量级替代方法 64
4.1.4 使用备用交换器处理无法路由的消息 66
4.1.5 基于事务的批量处理 68
4.1.6 使用HA队列避免节点故障 70
4.1.7 HA队列与事务 72
4.1.8 通过设置delivery-mode为2将消息持久化到磁盘 72
4.2 RabbitMQ回推 75
4.2.1 使用rabbitpy来检测连接状态 77
4.2.2 使用管理API管理连接状态 77
4.3 小结 78

第5章 消费消息,避免拉取 79

5.1 对比Basic.Get 和Basic.Consume 80
5.1.1 Basic.Get 80
5.1.2 Basic.Consume 82
5.2 优化消费者性能 84
5.2.1 使用no-ack模式实现更快的吞吐量 85
5.2.2 通过服务质量设置控制消费者预取 86
5.2.3 消费者使用事务 89
5.3 拒绝消息 90
5.3.1 Basic.Reject 90
5.3.2 Basic.Nack 91
5.3.3 死信交换器 92
5.4 控制队列 94
5.4.1 临时队列 94

5.4.2 永久队列 97
5.4.3 任意队列设置 99
5.5 小结 99

第6章 消息路由模式 101

6.1 通过direct交换器路由消息 102
6.1.1 创建应用架构 103
6.1.2 创建RPC工作者 107
6.1.3 编写简单的RPC发布者 110
6.2 通过fanout交换器广播消息 115
6.2.1 修改面部检测消费者 116
6.2.2 创建一个简单的图片哈希消费者 117
6.3 使用topic交换器有选择地路由消息 119
6.4 使用headers交换器有选择地路由消息 122
6.5 交换器性能基准 124
6.6 交换器间路由 125
6.7 使用一致性哈希交换器路由消息 127
6.8 小结 131

第二篇 管理数据中心或云中的RabbitMQ

第7章 RabbitMQ集群 135

7.1 集群简介 136
7.1.1 集群和管理界面 137
7.1.2 集群节点类型 138
7.1.3 集群和队列行为 139
7.2 集群设置 142
7.2.1 虚拟机设置 143
7.2.2 向集群中添加节点 144
7.3 小结 147

第8章 跨集群的消息分发 148
8.1 联合交换器和联合队列 149
8.1.1 联合交换器 149
8.1.2 联合队列 152
8.2 创建RabbitMQ虚拟机 153
8.2.1 创建首个实例 153
8.2.2 复制EC2实例 159
8.3 连接上游节点 162
8.3.1 定义联合中的上游节点 162
8.3.2 定义策略 164
8.3.3 利用上游集合 167
8.3.4 双向联合交换器 170
8.3.5 使用联合来升级集群 171
8.4 小结 173

第三篇 集成与定制

第 9 章 使用替代协议 177
9.1 MQTT和RabbitMQ 178
9.1.1 MQTT协议 178
9.1.2 通过MQTT发送消息 182
9.1.3 MQTT订阅者 184
9.1.4 MQTT插件配置 187
9.2 STOMP和RabbitMQ 189
9.2.1 STOMP协议 190
9.2.2 发布消息 191
9.2.3 消费消息 195
9.2.4 配置STOMP插件 198
9.2.5 在Web浏览器中使用STOMP 199
9.3 通过HTTP进行无状态发布 200
9.3.1 statelessd的由来 200

9.3.2 使用statelessd 201
9.3.3 运营架构 202
9.3.4 通过statelessd来发布消息 203
9.4 小结 203

第10章 数据库集成 205

10.1 PostgreSQL扩展：pg_amqp 206
10.1.1 安装pg_amqp扩展 207
10.1.2 配置pg_amqp扩展 209
10.1.3 通过pg_amqp发送消息 210
10.2 监听PostgreSQL通知 212
10.2.1 安装PostgreSQL LISTEN交换器 213
10.2.2 基于策略的配置 215
10.2.3 创建交换器 217
10.2.4 创建并绑定测试队列 217
10.2.5 通过NOTIFY发送消息 218
10.3 将消息存入InfluxDB中 219
10.3.1 InfluxDB的安装与设置 220
10.3.2 安装InfluxDB存储交换器 222
10.3.3 创建测试交换器 223
10.3.4 测试交换器 224
10.4 小结 227

附录 准备就绪 228

A.1 安装VirtualBox 228
A.2 安装Vagrant 230
A.3 设置Vagrant虚拟机 233
A.4 确认安装 234
A.5 小结 236

序 言

当 Manning Publications 于 2012 年 4 月发布《RabbitMQ 实战》一书时，RabbitMQ 迅速普及。时至今日，RabbitMQ 已然成为消息代理服务器世界的领导者之一，并且是各种应用程序应用场景的理想之选。通过分布式应用程序助力通信，在面向服务体系结构中采用微服务，并实现 CQRS 和事件源组件的逻辑分离，这都是一些常见的 RabbitMQ 用法。

通过对 AMQP 协议结构的研究，对各种交换器的逐步探索，以及对性能方面的考察等事项，我们以全新的视角对 RabbitMQ 本身进行深入探索。《深入 RabbitMQ》一书旨在将你对 RabbitMQ 的理解提升至新的水平，使你能在实际应用中进一步应用这些原理。

致 谢

本书写作并非一蹴而就。首先要感谢我们的家人和朋友，他们孜孜不倦地陪伴在左右，毫无怨言，特别是傍晚的那些咖啡让我们坚持不懈地编写这样一本书。再次说声感谢！

感谢《RabbitMQ 实战》（2012 年 4 月由 Manning Publications 出版）的作者 Alvaro Videla 和 Jason J.W. Williams，是他们激发了无数开发者对 RabbitMQ 的兴趣、开拓了大家的眼界，为日后的研究奠定了基础。

感谢我们的开发编辑 Karen，正是她在这段时间里付出的无尽耐心和理解，再加上整个 Manning 团队所有人的巨大努力，才将我们推向成功。这份艰苦卓绝的工作历经反复打磨，最终成就了本书，我们心存感恩！

同样感谢技术校对者 Karsten Strøbæk，他对本书的贡献非常大。还要感谢以下审稿人，他们的意见对本书的帮助同样重要。他们是 Phillip Warner、Jerry Kuch、Nadia Saad Noori、Bruce Snyder、Robert Kielty、Milos Milivojevic、Arathi Maddula、Ian Dallas、George Harley、Dimitri Aivaliotis、Hechen Gao、Stefan Turalski、Andrew Meredith、Artem Dayneko、David Paccoud、Barry Alexander、Biju Kunjummen、Adolfo Pérez Álvarez、Brandon Wilhite、David Pull 和 Ray Lugo。

还有许多朋友也以不同的方式为本书做出了贡献。我们无法提及每个人的名字，否则整本书都可能装不下。总之，非常感谢那些曾经帮助我们成就这本书的朋友们！

关于本书

RabbitMQ 是采用 Erlang 编写的开源消息代理服务器，目前隶属于 Pivotal Software。它基于 AMQP 开放协议，官方客户端库提供了基于 Java、.NET 和 Erlang 版本，以及大多数其他流行的编程语言编写的库。

本书力求和最新版的 RabbitMQ 3.6.3 保持一致。由于 RabbitMQ 本身的发布计划并不确定，在你拿到本书时 RabbitMQ 很可能已经发布了新的版本。别担心，根据我们的经验，RabbitMQ 极少会在新版本中削减特性，而只会增加新特性并修复问题。

本书中使用的代码示例是采用 Python 编写的。如果你没有安装 Python 和 RabbitMQ，或者你只是想做实验而不想安装完整的环境的话，我们编写了有关如何配置安装 Vagrant box 的教程，里面预装了所有需要的程序。请确保你查看了附录，清楚如何安装并运行。

路线图

第 1 章讨论了 RabbitMQ 的基础：RabbitMQ 众多的功能以及 RabbitMQ 的基础，高级消息通信队列模型。

第 2 章探讨了 AMQ 协议，讲解了帧结构，以及从 RabbitMQ 收发消息时底层处理逻辑。

第 3 章的内容更为深入，讲解了消息属性，包括消息头以及如何在应用程序中使用它们。我们可以利用消息头向消息添加重要的元数据信息，例如 content-type 以及编码类型。

第 4 章重点讲解了性能权衡。每种级别的保证都会对应用程序的性能产生影响。本章将介绍这些参数并帮助你在保证消息可靠性的环境所需与快速消息投递之间达到平衡，这就是所谓的金发姑娘原则（Goldilocks Principle）。

第 5 章探索了消息消费的概念，讲解了 Basic.Get 和 Basic.Consume 两者之间在底层上根本上的差异，以及通常情况下为什么使用 Basic.Consume 会更好。本章还讲解了消息预取（prefetching）、服务质量、消息确认、死信交换器、临时队列以及消息过期。

第 6 章深入讲解了 RabbitMQ 中的四种核心交换器类型，以及如何在应用架构中选择使用，并从中受益。

第 7 章着眼于如何通过集群管理来扩展 RabbitMQ，集群中的节点故障恢复，以及在应用集群环境时更进一步的性能考虑。

第 8 章的主要内容是集群的核心概念，包括互联（federate）交换器和队列，将 RabbitMQ 集群与 AWS 进行集成，以及各种策略的应用。

第 9 章讲解了与 RabbitMQ 通信的其他几种方式：使用 MQTT 和 STOMP 等替代协议，或者基于 statelessd 的 HTTP 协议进行消息通信。

最后，第 10 章介绍了数据库集成方面的内容，讲解了将 PostgreSQL、InfluxDB 与 RabbitMQ 集成的方式。

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在下载资源处下载。
- **提交勘误**：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34180

第一篇

RabbitMQ 和应用程序体系结构

在本书第一篇的内容中，我们将探讨 AMQ 协议的结构。它描述了应用程序是如何与 RabbitMQ 进行通信的。我们还会深入消息本身，利用消息头、优先级等功能来增强消息交互。我们将探索性能权衡，在稳定性、事务安全性与不确保这些因素的高性能吞吐量者之间达成平衡。另外，我们将研究不同的交换器类型以及它们各自的运作方式。

第 1 章　RabbitMQ 基础

本章概要：

- RabbitMQ 特性
- 为什么 RabbitMQ 正在成为消息通信架构的一种主流选择?
- 高级消息队列（Advanced Messaging Queuing）模型的基本要素，该模型构成了 RabbitMQ 的基础

无论你的应用是位于云端还是位于自身的数据中心，RabbitMQ 是一种轻量级的、功能非常强大的工具，可用来构建十分简单抑或异常复杂的分布式软件架构。作为一款消息中间件，在本章中你将学习 RabbitMQ 在处理和解决问题时如何为你提供强大的灵活性。你将了解一些公司对 RabbitMQ 的使用方式，同时掌握 RabbitMQ 的关键特性，正是这些关键特性促使 RabbitMQ 成为时下最流行的消息代理服务器（message broker）之一。

1.1 RabbitMQ 特性以及好处

RabbitMQ 拥有众多特性和好处，我们把最重要的列举如下：

- 开源——RabbitMQ 最初是由 LShift、LTD 和 Cohesive FT 三家合伙的 RabbitMQ 科技公司联合开发的，目前归属于 Pivotal 软件公司，并且以 Mozilla 公共协议进行发布。作为一款由 Erlang 语言开发的开源软件项目，RabbitMQ 具有自由度以及灵活性，而作为一款产品其背后则有 Pivotal 的大力支撑。活跃在 RabbitMQ 社区的开发者和工程师们能够贡献强化特性和附加功能，而 Pivotal 则提供商业支持以及作为产品持续发展的坚强后盾。
- 平台和供应商无关性——作为实现了具有平台和供应商无关性的高级消息队列协议（Advanced Message Queuing Protocol，AMQP）规范的一种消息代理服务器，RabbitMQ 为几乎全部开发语言提供了客户端工具并能运行在所有主流计算机平台上。
- 轻量级——RabbitMQ 是轻量级的，运行 RabbitMQ 核心功能以及诸如管理界面的插件只需要不到 40MB 内存。请注意往队列中添加消息可能会增加其内存使用量。
- 面向大多数现代语言的客户端开发库——作为代理服务器，RabbitMQ 提供了一种引人注目的编程模式，其客户端开发库的目标是支持绝大多数编程语言并能运行于多个平台。当你通过编程与 RabbitMQ 进行通信时，不受任何供应商或开发语言的限制。事实上，当由不同语言实现的应用程序之间需要交互时，把 RabbitMQ 当作核心组件的场景并不少见。RabbitMQ 为不同的开发语言之间进行跨操作系统和环境的数据共享提供了一种有用的桥梁，这些语言包括 Java、Ruby、Python、PHP、JavaScript 和 C# 等。
- 灵活控制消息通信的平衡性——在消息吞吐量和性能上，RabbitMQ 提供了一种灵活性用于控制这两者在可靠消息通信上的平衡。因为它并不是一种“适合所有场景”的应用，消息在投递之前可以指定为是否持久化到硬盘；如果在一个集群中，队列可以被设置成高可用，即消息会存储在多台服务器中确保在某台服务器宕机时不会丢失。
- 高延迟性环境插件——因为不是所有的网络拓扑和架构都是一样的，RabbitMQ 既支持在低延迟环境下的消息通信机制，也提供了针对如互联网的高延迟环境下的插件。这就使得 RabbitMQ 可以在同一个本地网络或者在跨越多个数据中心的共享互联（federated）机制下构建消息集群。
- 第三方插件——作为应用集成的一个关键要素，RabbitMQ 提供了灵活的插件系统。例如，当需要 RabbitMQ 进行数据库直接写入时，就可以使用第三方插件把消息直

接存储到数据库中。

- 多层安全——在 RabbitMQ 的多个层次中包含着安全性设计。客户端连接可以通过使用 SSL 通信和客户端证书验证以提高安全性。在虚拟主机（virtual-host）层可以管理用户访问，从而在较高层次实现消息和资源的隔离。另外，通过配置可以使用正则表达式模式匹配的方式控制从队列中读取消息和把消息写入交换器的过程。最后，使用插件可与类似 LDAP 的外部认证系统进行集成。

本章后续内容会对上述所列功能进行详细阐述，但在这里我想先强调 RabbitMQ 中的两个最基本的特性：实现语言（Erlang）和所基于的模型（高级消息队列模型），该模型提供了定义 RabbitMQ 中大多数专业术语和行为的规范。

1.1.1　RabbitMQ 与 Erlang

作为一款高性能、稳定且支持集群化的消息代理服务器，RabbitMQ 无疑是构建大规模消息架构的核心组件，并在一些重要应用环境中占有一席之地。RabbitMQ 基于 Erlang 语言开发，该语言是 20 世纪 80 年代中后期由 Ericsson 计算机科学实验室设计的面向电信行业的函数式编程语言。Erlang 被设计成一种分布式、高容错的软实时系统，用于构建 99.999% 可用性的应用系统。作为一种开发语言和运行时系统，Erlang 专注于节点之间消息通信的轻量级进程，提供了状态无关的高并发性。

实时系统　实时系统（Real-Time System）可以是一种硬件平台、软件平台，或者两者兼有，用于满足针对某个事件必须返回响应的需求。软实时系统则是指在任务执行过程中，为了那些更重要的任务会牺牲部分非重要任务的时效性。

基于并行处理和消息通信设计，Erlang 成为构建类似 RabbitMQ 的消息代理服务器的自然选择：消息代理服务器作为一种应用程序，维护并发连接、实现消息路由（route）并管理它们的状态。同时，Erlang 的分布式通信架构天然可以用于构建 RabbitMQ 集群机制。RabbitMQ 集群中的服务器充分利用 Erlang 的进程间通信（*Inter-Process Communication*，IPC）系统，具备其他竞品消息代理服务器不得不去实现的集群功能（见图 1.1）。

尽管 Erlang 给 RabbitMQ 带来了很多优势，但 Erlang 环境却可能成为一种绊脚石。学习 Erlang 相关知识是有帮助的，可以确保你有信心去管理 RabbitMQ 配置，以及明确如何使用 Erlang 去收集关于 RabbitMQ 当前运行时的状态信息。

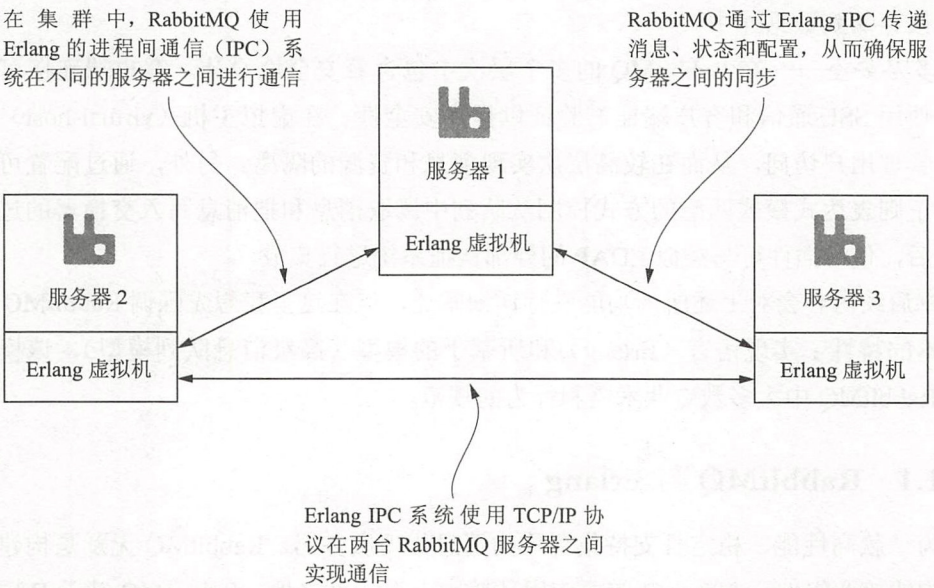


图 1.1　RabbitMQ 集群使用 Erlang 自带的进程间通信机制实现虚拟机之间的跨节点通信、共享状态信息以及允许整个集群内进行消息的发布和消费

1.1.2　RabbitMQ 与 AMQP

RabbitMQ 首发于 2007 年，互操作性、性能和稳定性是其主要的实现目标。同时，RabbitMQ 也是最早实现 AMQP 规范的消息代理服务器之一。从各个方面来看，它都是一种参考实现。AMQP 规范可以拆分成两部分，不仅定义了与 RabbitMQ 交互的点对点协议，也提供了描述 RabbitMQ 核心功能的逻辑模型。

注意：AMQP 规范存在多个版本。基于本书的写作目的，我们只关注于 AMQP 0-9-1。尽管较新版本的 RabbitMQ 提供了一个扩展插件以支持 AMQP 0-10，但 RabbitMQ 内核架构还是比较接近于 AMQP 0-8 和 AMQP 0-9-1 的。AMQP 规范主要由两份文档组成，一份是描述 AMQ 模型和 AMQ 协议的高层次文档；另一份则更为详细，提供了关于每个类、方法、属性和字段的多层次的详细信息。关于 AMQP 规范文档的更多信息，可参考 http://www.amqp.org。

市面上存在多种流行的消息代理服务器和消息通信协议，对你而言重要的是需要考虑具体协议和代理服务器对自身应用的影响。RabbitMQ 在支持 AMQP 的同时，也支持其他诸如 MQTT、Stomp 和 XMPP 等协议。与其他流行的消息代理服务器相比，RabbitMQ 的协议中立性和插件扩展性使其成为构建多协议应用架构的明智选择。

RabbitMQ 基于 AMQP 规范，该规范概括了 RabbitMQ 的主要架构和通信模式。当我们对 RabbitMQ 和其他消息代理服务器进行评估时，这是一个重要区别。通过 AMQP，RabbitMQ 致力于成为一个与供应商无关、平台独立的解决方案，用于满足面向消息通信架构所需满足的复杂需求，这些需求包括灵活的消息路由、配置化的消息持久化以及跨数据中心通信。

1.2　谁在使用 RabbitMQ，在怎么用

作为一款开源的软件包，RabbitMQ 迅速得到广泛的应用，并在一些规模、流量最大的互联网网站上发挥作用。众所周知，现在 RabbitMQ 运行在很多不同的环境以及很多不同类型的公司和组织中：

- Reddit，一家流行的在线社区，每个月在它们的应用平台核心功能上充分利用 RabbitMQ 服务于 10 亿级别的网页。当用户进行网上注册、提交一个新的公告或者对一个链接进行投票时，一条消息就会被发送到 RabbitMQ 并被消费系统异步处理。
- NASA 在它们的 Nebula 平台中选择 RabbitMQ 作为消息代理服务器，Nebula 为 NASA 的服务器基础设施提供集中式服务器管理平台，该服务器基础设施正成长为 OpenStack 平台，后者则是一个非常流行的、用于构建私有和公有云服务的软件平台。
- 在 Agoura Games，RabbitMQ 在面向社区的在线游戏平台中处于核心位置，对大量实时单人和多人游戏数据和事件进行路由。
- 对于 Ocean Observations Initiative 而言，RabbitMQ 将关键的物理、化学、地理和生物数据路由到分布式计算机网络中。从南太平洋和大西洋的传感器中收集的数据是一项国家科学基金项目的组成部分，该项目涉及构建一个分布于海洋和海底的大规模传感器网络。
- Rapportive，一个用于将合同详细信息放在收件箱中的 Gmail 附加功能，使用 RabbitMQ 作为数据处理系统的黏合剂。每个月有数以十亿计的消息通过 RabbitMQ 进行传递，为 Rapportive 的网页抓取引擎和分析系统提供数据，从而去除需要在 Web 服务器上进行长时间运行的操作。
- MercadoLibre，拉丁美洲最大的电子商务生态系统，在它们的企业服务总线（Enterprise Service Bus，ESB）架构的核心部分使用 RabbitMQ 对来自紧耦合应用的数据进行解耦，从而确保应用架构中的各种组件能够灵活集成。

- Google 的 AdMob 移动广告网络在它们的 RockSteady 项目中使用 RabbitMQ 实现实时度量分析与故障检测，通过构建一个过滤漏斗把消息经由 RabbitMQ 流转到复杂事件处理系统 Esper 中。
- 印度生物识别数据库系统 Aandhaar 利用 RabbitMQ 处理工作流中不同阶段的数据，并把数据投递到它们的监控工具、数据仓库以及基于 Hadoop 的数据处理系统中。Aandhaar 是为每一个印度居民提供在线便携式身份认证的一个系统，服务于 12 亿用户。

正如你所看到的，RabbitMQ 不仅仅用于一些大型的互联网站点，在学术界它也可以用于大规模的科学研究，NASA 发现 RabbitMQ 适合用于网络基础设施管理栈的核心部分。正如这些例子所示，RabbitMQ 已经被应用于很多不同环境和产业下的关键系统中并取得了巨大成功。

1.3 松耦合架构的优势

当我第一次开始实现一个基于消息通信的架构时，我在为用户登录网站的场景寻找一种实现方法以便解耦相应的数据库更新操作。网站成长得非常快，但基于现有的设计方案无法实现良好的扩展性。当一个会员成功登录到网站时，几台数据库服务器中的表需要去更新登录时间戳（见图 1.2）。该时间戳需要实时更新，因为网站上很多相关活动在某种程度上受这个时间戳值驱动。在社交游戏中，刚登录成功的会员会比其他任何时候都在线的用户处于优先状态。

网站不断发展，会员登录的时间也随之增加。原因显而易见，当添加一个依赖于会员最近登录时间戳的应用时，为了提供尽可能快的响应速度，该应用中的数据库表将保存这个时间戳以避免跨库联合查询。为了确保数据的实时性和准确性，每当会员登录时，新增应用中的数据库表也需要同时更新。采用这种方式，在数据库表不是太多的情况下处理时间并不会太长。性能问题逐渐滋生的原因在于这些数据库更新采用的是串行方式。每一个更新会员最新登录时间戳的查询操作都需要前一个操作完成之后才能开始。如果有 10 个性能不错的查询操作，每个操作在 50ms 之间完成，那么累加起来仅仅数据库更新就需要半秒时间。所有这些查询操作都需要在发送授权响应以及重定向到用户之前完成。此外，任何数据库服务器操作上的纰漏都会加剧这一问题。如果一台数据库服务器响应变慢或者变成不可用，会员将无法成功登录网站。

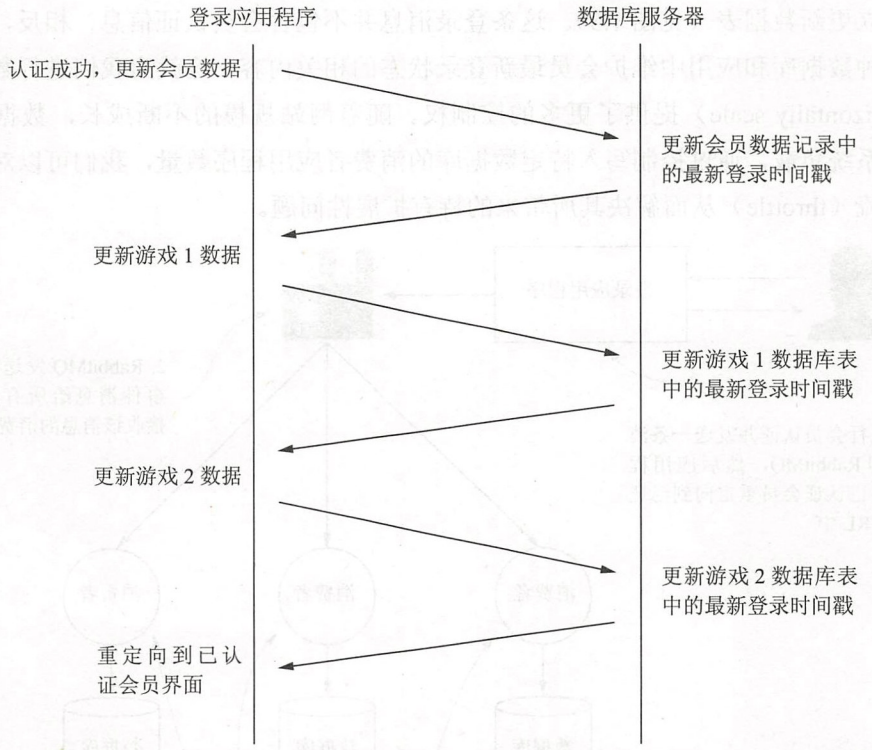


图 1.2　修改前：每当用户登录，每个数据库都将以顺序和相互依赖的方式更新时间戳。所添加的表越多，所需时间也就越长

为了解耦面向用户的登录应用和数据库写操作，我将视角投向往消息中间件或集中式消息代理服务器中发送消息，它们可以把消息分散到任意数量的消费者应用程序中，然后由这些消费者应用程序执行数据库写操作。通过对几款不同消息代理服务器进行试验，最终我选择了 RabbitMQ。

定义　消息中间件（Message-oriented middleware，MOM）是一种软件或硬件基础设施，通过它可以在分布式系统中发送和接收消息。RabbitMQ 通过高级路由和消息分发功能巧妙地实现了这一角色，即使需要满足广域网（Wide Area Network，WAN）环境下实现可靠性所应达到的容错条件，分布式系统也可以很容易与其他系统进行互连。

在将登录过程从所需的数据库更新中解耦之后，我发现了一种新的自由度。会员能够快速登录，是因为我们已经不把数据库更新作为整个认证过程中的一环。取而代之的是发送一条会员登录消息，该消息包含了数据库更新所需的相关内容，而消费者应用程序则被设计

成可以独立更新数据表（见图 1.3）。这条登录消息并不包含会员认证信息，相反，它只包含需要在各种数据库和应用中维护会员最新登录状态的相关内容。这就为我们进行数据库水平扩展（horizontally scale）提供了更多的控制权。随着网站规模的不断成长，数据库写入操作加重了系统负载，通过控制写入特定数据库的消费者应用程序数量，我们可以对数据库操作进行限流（throttle）从而解决其所带来的特有扩展性问题。

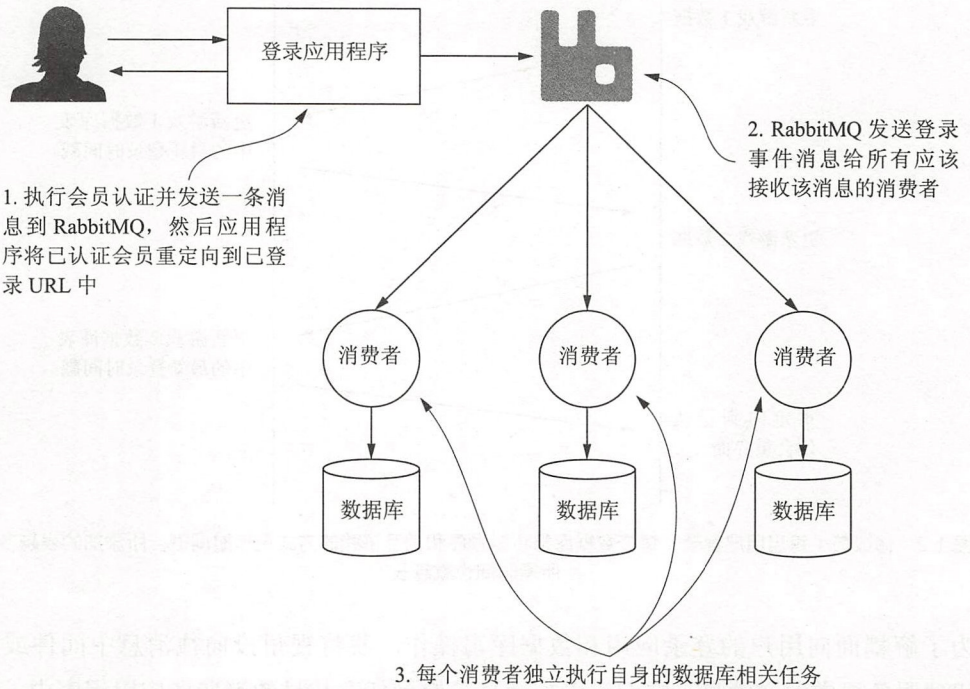


图 1.3　修改后：通过使用 RabbitMQ，松耦合数据被异步、独立地发布到各个数据库，从而使登录应用的处理过程不需要等待任何数据库写入

我详细介绍了基于消息通信架构的优势，但有一点也非常重要，即这些优势同样可能影响系统的性能，正如上文中描述的登录架构。有很多问题可能影响发布者性能，包括网络问题以及 RabbitMQ 对消息发布者进行限流。当这些事件发生时，你的应用性能将会下降。除了对消费者进行水平扩展之外，制定水平扩展消息代理服务器的计划，以便获取更好的消息吞吐量和发布者性能，也是一个明智之举。

1.3.1　解耦你的应用

对于那些寻求以数据为中心的灵活应用架构的组织而言，使用消息中间件可以获取极大的优势。通过向基于 RabbitMQ 的松耦合设计迁移，应用架构不再受限于数据库写入的性

能瓶颈，并且可以在不需要改动其他任何核心应用的前提下就可以轻松添加新的应用来操作数据。图 1.4 展示了紧耦合应用设计，这种设计通过数据库进行交互。

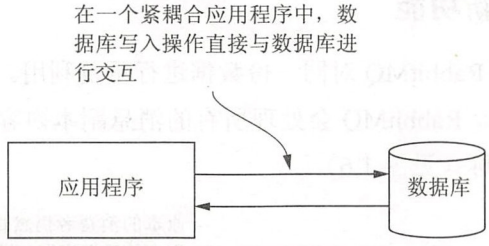


图 1.4　当与数据库进行交互时，紧耦合应用必须等待数据库服务器的响应才能进行下一步操作

1.3.2　解耦数据库写入

紧耦合架构中，应用在完成一个事务之前必须等待数据库服务器的响应。这样的设计无论在同步还是异步应用中都存在潜在的性能瓶颈。一旦数据库服务器因为缺少优化或硬件问题而出现性能下降，那么应用响应速度也会随之变慢。如果数据库停止响应或宕机，那么应用也可能发生宕机。

通过将数据库从应用中解耦出来，就可以创建一种松耦合架构。在这种架构中，作为消息中间件的 RabbitMQ 扮演着一个中介角色，即处理入库操作之前的数据。消费者从 RabbitMQ 服务器中获取数据，然后执行与数据库相关的操作（见图 1.5）。

在这个模型中，如果数据库需要进行离线维护，或者写入负载量变得太大，你就可以对消费者应用程序实行限流或者直接关闭。在消费者能够接收消息之前，这些数据都会保存

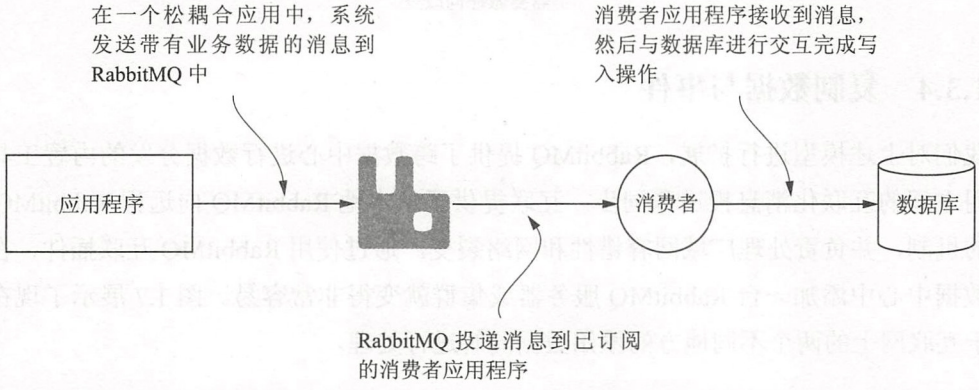


图 1.5　松耦合应用使得原本把数据直接存在数据库中的应用能够把数据发送到 RabbitMQ，从而实现异步数据处理

在队列中。这种暂停或限流消费者应用程序的行为恰恰就是使用这类架构的一种优势。

1.3.3 无缝添加新功能

松耦合架构同样允许 RabbitMQ 对同一份数据进行重复利用。原本只被写到数据库中的数据可以被用作其他目的。RabbitMQ 会处理所有的消息副本内容，并将它们路由到多个消费者以满足不同的处理目标（见图 1.6）。

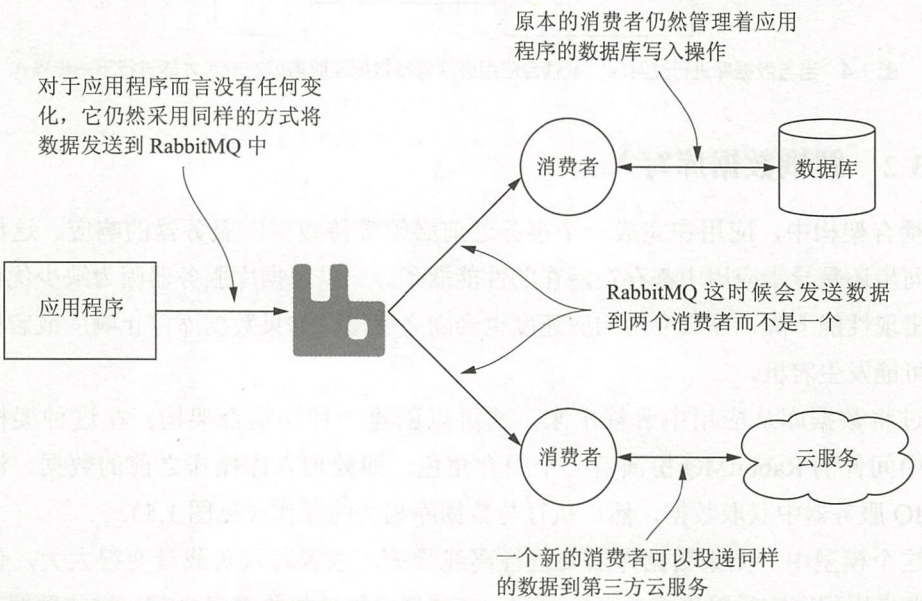


图 1.6 通过使用 RabbitMQ，当将同一份数据同时发送到新的云服务和原本的数据库时，发布者应用程序不需要做任何改变

1.3.4 复制数据与事件

我们对上述模型进行扩展，RabbitMQ 提供了跨数据中心进行数据分发的内置工具，支持应用之间的互联化消息投递和同步。互联提供了从本地 RabbitMQ 向远程 RabbitMQ 发送消息的机制，并负责处理广域网容错性和网络裂变。通过使用 RabbitMQ 互联插件，在另外一个数据中心中添加一台 RabbitMQ 服务器或集群就变得非常容易。图 1.7 展示了现在可以对位于互联网上的两个不同地方的原始数据同时进行处理。

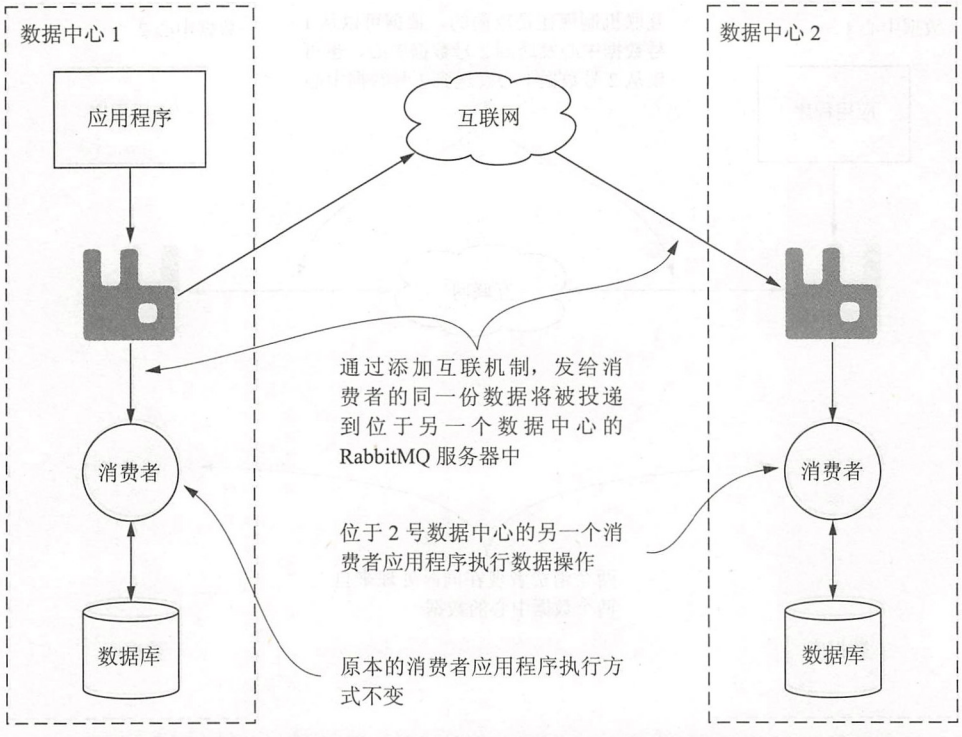


图 1.7　应用 RabbitMQ 的互联插件，消息可以在多个数据中心之间进行复制并执行相同的任务

1.3.5　多主（Multi-Master）互联化数据与事件

我们对上述概念做进一步扩展，在 2 号数据中心中，通过添加相同的前端应用，然后设置 RabbitMQ 服务器为双向接收互联数据，你就可以创建分布在不同物理位置的高可用应用。位于任意数据中心上的应用所产生的消息将被同时发送到两个数据中心中的消费者那里，从而实现在数据存储和处理上的冗余机制（见图 1.8）。应用架构的这一构建方式为应用提供了水平扩展（scale horizontally）和用户地域无关性，同时也为你的基础设施实现分布式处理提供了一种高性价比的解决方案。

注意　如同任何一种架构决策，消息中间件会引入一定程度的操作复杂性。因为消息代理服务器成为了应用设计中的一个中心节点，从而引入了新的单点故障。本书中我们会介绍一些策略用于创造高可用解决方案，从而最小化这一风险。另外，增加消息代理服务器也意味着创建了一个需要管理的新应用。在对架构中是否要引入消息代理服务器进行权衡时，必须要考虑配置、服务器资源以及监控问题。本书后续章节将阐述如何考虑这些问题以及其他关注点。

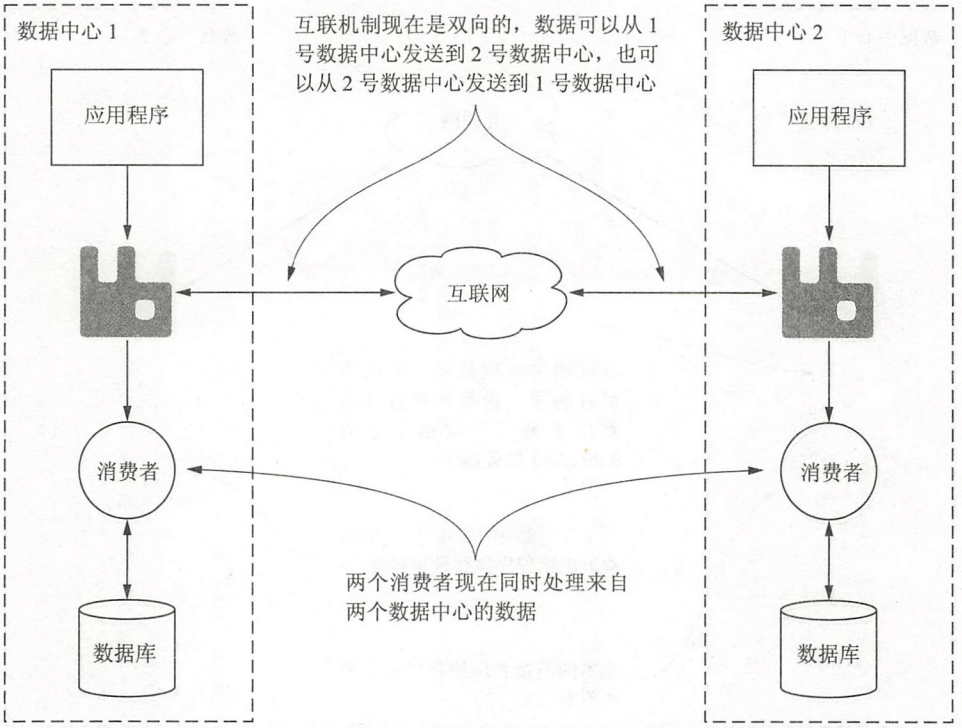


图 1.8　双向互联数据确保同一份数据事件能同时在两个数据中心进行接收和处理

1.3.6　高级消息队列模型

RabbitMQ 的很多强大功能和灵活性来自于 AMQP 规范。不像 HTTP 和 SMTP 协议，AMQP 规范不仅定义了一种网络协议，同时也定义了服务器端的服务和行为。这些信息就是高级消息队列（Advanced Message Queuing，AMQ）模型。针对代理服务器软件，AMQ 模型在逻辑上定义了三种抽象组件用于指定消息的路由行为：

- 交换器（*Exchange*），消息代理服务器中用于把消息路由到队列的组件。
- 队列（*Queue*），用来存储消息的数据结构，位于硬盘或内存中。
- 绑定（*Binding*），一套规则，用于告诉交换器消息应该被存储到哪个队列。

RabbitMQ 的灵活性来自于消息如何通过交换器路由到队列的动态特性。介于交换器和队列之间的绑定，以及它们所创建的动态消息路由，构成了消息通信架构的基本组件。使用 RabbitMQ 所提供的这些基础工具来创建正确的结构，可以使你的应用具有扩展性，并能轻松应对业务需求变更。

为了把消息路由到合适的目标地址，RabbitMQ 所需的第一种信息就是用于控制路由的交换器。

交换器

交换器是 AMQ 模型定义的三种组件之一。一个交换器接收发送到 RabbitMQ 中的消息并决定把它们投递到何处。交换器定义消息的路由行为，通常这需要检查消息所携带的数据特性或者包含在消息体内的各种属性。

RabbitMQ 拥有多种交换器类型，每一种类型具备不同的路由行为。另外，它还提供了一种可用于自定义交换器的插件架构。图 1.9 展示了发布者发布消息到 RabbitMQ 中，然后通过交换器进行消息路由的逻辑视图。

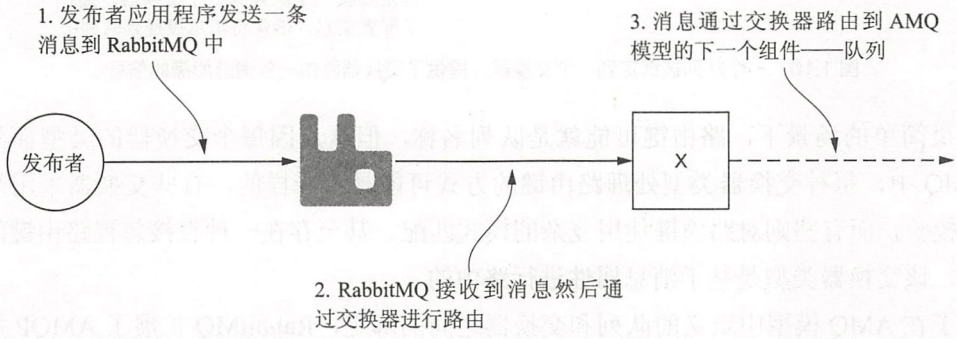


图 1.9　当发布者发布消息到 RabbitMQ 时，消息首先到达的是一个交换器

队列

队列负责存储接收到的消息，同时也可能包含如何处理消息的配置信息。队列可以把消息只存储在内存中，也可以存储在硬盘中，然后以先进先出（FIFO）的顺序进行投递。

绑定

AMQ 模型使用绑定（*binding*）来定义队列和交换器之间的关系。在 RabbitMQ 中，绑定或绑定键（*binding-key*）即告知一个交换器应该将消息投递到哪些队列中。对于某些交换器类型，绑定同时告知交换器如何对消息进行过滤从而决定能够投递到队列的消息。

当发布一条消息到交换器时，应用程序使用路由键（*routing*-key）属性。路由键可以是队列名称，也可以是一串用于描述消息、具有特定语法的字符串。当交换器对一条消息进行评估以决定路由到哪些合适的队列时，消息的路由键就会和绑定键进行比对（见图 1.10）。

换句话说，绑定键是绑定队列到交换器的粘合剂，而路由键则是用于比对的标准。

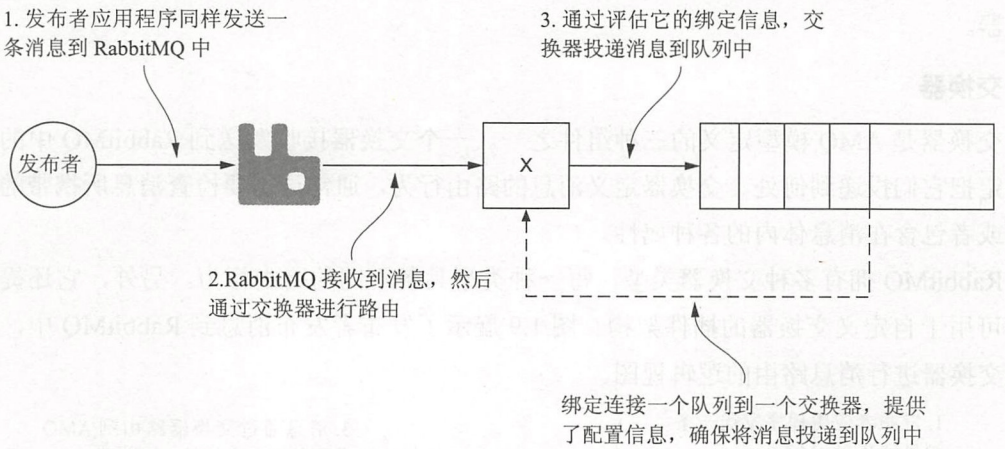


图 1.10　一个队列被绑定到一个交换器，提供了交换器路由一条消息所需的信息

在最简单的场景下，路由键可能就是队列名称，但这点因每个交换器的类型而异。在 RabbitMQ 中，每种交换器类型处理路由键的方式可能是不一样的，有些交换器采用简单的相等性校验，而有些则对路由键使用复杂的模式匹配。甚至存在一种直接忽视路由键的交换器类型，该交换器类型是基于消息属性进行路由的。

除了在 AMQ 模型中定义的队列和交换器之间的绑定，RabbitMQ 扩展了 AMQP 规范并提供了交换器绑定交换器的功能。这种特性为创建不同消息路由模式提供了强大的灵活性。除了基于交换器的各种可用路由模式，你将在本书第 6 章中学习更多关于交换器绑定交换器的内容。

1.4　小结

作为消息中间件，RabbitMQ 是一项令人振奋的技术，它提供一种操作灵活性，这种灵活性在不基于它所提供的松耦合架构下是很难实现的。通过深入分析 AMQP 基础及行为，本书将是一份有用的参考资料，为你提供应用程序如何充分利用 RabbitMQ 健壮性和强大功能的真知灼见。尤其是，你将马上就能学习如何通过 RabbitMQ 发送消息，以及如何使用动态路由功能选择性地获取应用程序可以发送的数据，这些数据在你的环境中可能隐藏在紧耦合的代码和流程深处。

不管你是一位应用程序开发者还是一位高级应用架构师，深入了解 RabbitMQ 的多样化功能特性是有帮助的，因为你的应用程序将从这些功能特性中获益。到目前为止，你已经学习了构成 AMQ 模型的最基本概念。本书第一部分将对这些概念进行扩展，你将学习 AMQP 以及它如何定义 RabbitMQ 的核心行为。

因为这是一本面向实践的书，其目的是提供典型场景下使用 RabbitMQ 所需的知识，所以你将在下一章开始使用代码。通过学习“如何与 Rabbit 交互”，你将利用 AMQP 的基本概念去编码实现如何通过 RabbitMQ 发送和接收消息。为了与 Rabbit 进行交互，你将使用一款叫作 *rabbitpy* 的第三方库，该第三方库基于 Python 实现，本书将使用这款第三方库编写示例代码，我们会在下一章中介绍它。即使你是一位经验丰富的开发人员，已经编写过与 RabbitMQ 交互的应用程序，你也至少应该对下一章做大体浏览，了解使用 RabbitMQ 时在 AMQP 协议层面具体发生了什么。

第 2 章　使用 AMQ 协议与 Rabbit 进行交互

本章概要：

- 使用 AMQ 协议与 RabbitMQ 进行通信
- AMQ 协议的低层帧结构
- 向 RabbitMQ 发布消息
- 从 RabbitMQ 消费消息

RabbitMQ 和客户端库进行交互的主要目的是完成你的应用程序与 RabbitMQ 之间的消息发布和消费，这些交互过程可能非常复杂。如果你处理的是一些诸如销售数据的重要信息，那么对销售有关的重要信息来源进行可靠投递应该是重中之重。客户端和代理服务器之间需要进行协商和通信以实现信息的传递，AMQP 规范在协议的层面上为这一过程定义了语义。通常，在 AMQP 规范中定义的专业术语也会体现在 RabbitMQ 的客户端库中，RabbitMQ 通信时用到的类和方法与协议层面的类和方法一一对应。理解这种通信的运行过程不仅能够帮助你掌握通信是如何发生的（how），还能够让你掌握其背后的原理（why）。

尽管客户端库中的命令倾向于模仿甚至直接复制 AMQP 规范中定义的操作，但多数客户端库都会尝试隐藏使用 AMQ 协议进行通信的复杂性。当你编写一个应用程序而不想过多考虑实现过程的复杂性时，这往往是件好事。但是当你想真正理解你的应用程序正在发生什么时，RabbitMQ 客户端背后的技术原理将不容忽视。不管是想去分析为什么应用程序发布消息比你预想的要慢，还是只想了解客户端与 RabbitMQ 第一次建立连接所需要的步骤，掌握客户端如何与 RabbitMQ 进行交互都能使上述过程更为容易。

为了更好地展示过程和原理，在本章中你将了解到 AMQP 把客户端和代理服务器之间的通信数据拆分成一种叫作帧（*frame*）的块结构，同时了解这些帧如何详细描述客户端应用程序和 RabbitMQ 相互之间所需要执行的操作。另外，你还将掌握如何在协议层面构建这些帧结构，以及它们如何为消息的投递和消费提供实现机制。

基于这些信息，你将使用为本书编写的 RabbitMQ 客户端库来实现第一个 Python 应用程序。该应用程序将使用 AMQP 来定义交换器和队列，然后将它们绑定在一起。最后，你将编写一个消费者应用程序，它将从新定义的队列中读取消息并打印消息内容。如果你已经熟悉这些内容，那么你也应该深入理解这一章内容。我发现直到我完全理解了 AMQP 的语义之后，对于 RabbitMQ 我才知道“why”，而不仅仅是“how”。

2.1　AMQP 作为一种 RPC 传输机制

作为一种 AMQP 代理服务器，RabbitMQ 提供了一套严格的通信方式，即在与核心产品进行通信的各个方面几乎都采用了远程过程调用（*Remote Procedure Call*，RPC）模式。远程过程调用是计算机之间的一种通信方式，它允许一台计算机在另一台计算机上执行一段程序或者一个方法。如果你已经经历过使用 Web 编程实现与远程 API 进行通信，那么你使用的就是一种常见的 RPC 模式。

然而，与 RabbitMQ 通信时发生的 RPC 会话与大多数基于 Web 的 API 调用不同。在大多数 Web API 定义中，RPC 会话发生在客户端发出命令并且服务器进行响应的过程中，服务器并不会向客户端发回命令。而在 AMQP 规范中，服务器和客户端都可以发出命令。对于客户端应用程序而言，这意味着它应该监听来自服务器的通信，这与客户端应用程序正在做的事情可能没什么关系。

为了说明客户端与 RabbitMQ 进行通信时 RPC 是如何工作的，我们来考虑一下连接协商过程。

2.1.1 启动会话

当你在国外与一位陌生人交流时，不可避免地会用一句问候语开始对话，让彼此之间能够知道对方是否能说同一种语言。当与 AMQP 交互时，这个问候语就是协议头（*protocol header*），由客户端发送给服务器。这个问候语不应该被认为是一个请求，但是与其余的会话不同，这并不是一个命令。RabbitMQ 通过 `Connection.Start` 命令响应问候语来启动命令/响应序列，而客户端则使用 `Connection.StartOk` 响应帧来响应 RPC 请求（见图 2.1）。

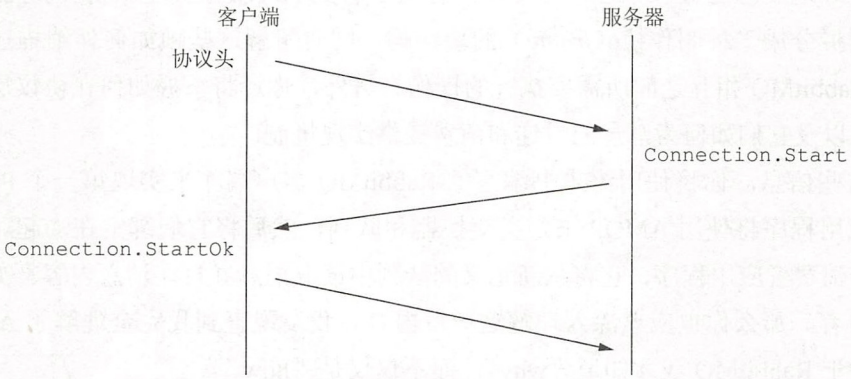


图 2.1　与 RabbitMQ 的初始通信协商演示了 AMQP 中的 RPC 过程

除非你正在编写一个客户端库，否则初始化连接的完整对话并不是非常重要，但值得注意的是，要完全连接到 RabbitMQ 需要完成由三个同步 RPC 请求所组成的序列，这三个 RPC 请求分别对应启动、调整和打开连接操作。这个序列一旦完成，RabbitMQ 就已经准备好为你的应用程序创建请求了。

应用程序可以发送给 RabbitMQ 的命令有很多种不同的类型，RabbitMQ 可以发送给客户端的命令也是一样。本章稍后将学习这些命令中的一小部分，但在此之前，你需要打开一个信道。

2.1.2 调整正确的信道

在概念上与双向无线电的信道类似，AMQP 规范定义了与 RabbitMQ 进行通信的信道。双向无线电使用无线电波作为它们之间的连接并传输信息。在 AMQP 中，信道使用协商的 AMQP 连接作为相互传输信息的渠道，而且它们将传输过程与其他正在进行中的会话隔离开来，这点也和双向无线电信道类似。一个 AMQP 连接可以有多个信道，允许客户端和服

务器之间进行多次会话。从技术上讲，这被称为多路复用（*multiplexing*），对于执行多个任务的多线程或异步应用程序来说，它非常有用。

提示　在创建客户端应用程序时，不要使用过多的信道使事情变得复杂。在编组帧的线路上，信道不过是分配给服务器和客户端之间所传递消息的一个整数值；而在 RabbitMQ 服务器和客户端中，它们代表更多的含义。因为会为每个信道设置内存结构和对象，连接中的信道越多，RabbitMQ 用于管理该连接的消息流所需的内存也就越多。如果你能合理地使用它们，你将会有一个更健康的 RabbitMQ 服务器和一个更简洁的客户端应用程序。

2.2　AMQP RPC 帧结构

与 C++、Java 和 Python 等语言中的面向对象编程概念非常相似，AMQP 使用类和方法在客户端和服务器之间创建公共语言，这些类和方法被称为 AMQP 命令（AMQP *commands*）。AMQP 中的类定义了一个功能范围，每个类都包含执行不同任务的方法。在连接协商过程中，RabbitMQ 服务器发送一个 `Connection.Start` 命令，然后编组成一个帧并发送给客户端。如图 2.2 所示，`Connection.Start` 命令由两个组件组成：AMQP 类（*Class*）和方法（*Method*）。

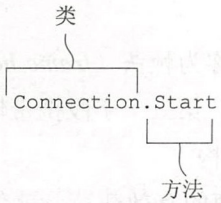


图 2.2　AMQP Connection 类以及 Start 方法构成了 Connection.Start RPC 请求

AMQP 规范中定义了很多命令，我相信你会和我一样想要跳过它们而直接去了解消息的发送与接收。然而，了解 RabbitMQ 中命令的网络传输形式，以便真正理解应用程序的运行过程，是非常重要的。

2.2.1　AMQP 帧组件

当使用命令与 RabbitMQ 进行交互时，执行这些命令所需的所有参数被封装在一个称为帧的数据结构中，帧对数据进行编码以便传输。帧为命令及其参数提供了一种有效的方式，用于在网络上进行编码和分隔。你可以把帧想象成一列火车上的货箱。概括来讲，货箱都具

有相同的基本结构，但是因所包含的内容而异。低层的 AMQP 帧也是如此。 如图 2.3 所示，低层 AMQP 帧由五个不同的组件组成：

- 帧类型
- 信道编号
- 以字节为单位的帧大小
- 帧有效载荷
- 结束字节标记（ASCII 值 206）

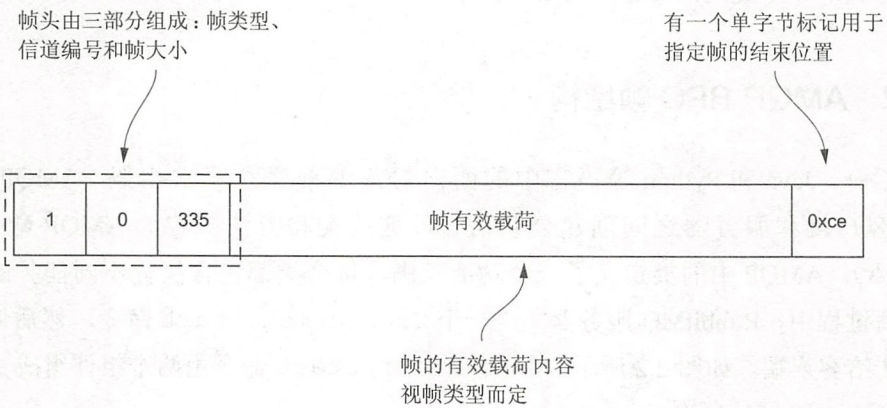


图 2.3　低层 AMQP 帧的构造

低层 AMQP 帧的头部是三个字段，这三个字段组合起来被称为帧头（*frame header*）。第一个字段是指示帧类型的单个字节，第二个字段指定帧的信道，第三个字段携带帧有效载荷的字节大小。帧头和结束字节标记合在一起创建了帧的基本结构。

在帧内部，位于头部之后和结束字节标记之前的内容就是帧的有效载荷。就像货车上的货箱能够保护自身内部物品一样，帧的设计也是为了保护其携带内容的完整性。

2.2.2　帧类型

AMQP 规范定义了五种类型的帧：协议头帧、方法帧、内容头帧、消息体帧及心跳帧。每种帧类型都有明确的目的，而有些帧的使用频率比其他的要高很多。

- 协议头帧用于连接到 RabbitMQ，仅使用一次。
- 方法帧携带发送给 RabbitMQ 或从 RabbitMQ 接收到的 RPC 请求或响应。
- 内容头帧包含一条消息的大小和属性。

- 消息体帧包含消息的内容。
- 心跳帧在客户端与 RabbitMQ 之间进行传递，作为一种校验机制确保连接的两端都可用且在正常工作。

在使用客户端库时，协议头帧和心跳帧对于开发者而言通常是透明的，而在编写与 RabbitMQ 进行通信的应用程序时，方法帧、内容头帧、消息体帧以及它们的结构通常是可见的。在下一节中，你将学习如何把与 RabbitMQ 进行交互的消息进行编组，并存放到方法帧、内容头帧以及一个或多个消息体帧中。

注意　AMQP 中的心跳行为用于确保客户端和服务器之间相互响应，这是展示 AMQP 作为一种双向 RPC 协议的完美示例。如果 RabbitMQ 发送心跳到你的客户端应用程序，然后没有得到响应，RabbitMQ 就会断开连接。通常情况下，单线程或异步开发环境下的开发人员会希望将超时时间设置为一个较大的值。如果你发现你的应用程序在一定程度上阻塞了通信，使得心跳机制难以正常运作，那么可以在创建客户端连接时将心跳间隔设置为 0 来关闭它们。相反，如果你选择使用比默认值 600 秒高得多的值，则可以通过更改 rabbitmq.config 文件中的 `heartbat` 值来更改 RabbitMQ 的最大心跳间隔。

2.2.3　将消息编组成帧

我们使用方法帧、内容头帧和消息体帧向 RabbitMQ 发布消息。发送的第一个帧是携带命令和执行它所需参数（如交换器和路由键）的方法帧。方法帧之后是内容帧，包含内容头和消息体。内容头帧包含消息属性以及消息体大小。AMQP 的帧大小有一个上限，如果消息体超过这个上限，消息内容将被拆分成多个消息体帧。这些帧始终以相同的顺序发送：方法帧、内容头帧以及一个或多个消息体帧（见图 2.4）。

如图 2.4 所示，当向 RabbitMQ 发送消息时，在方法帧中会发送一个 `Basic.Publish` 命令，随后是一个带有消息属性的内容头帧，该内容头帧包括消息的内容类型和发送时间等。这些属性被封装在 AMQP 规范中所定义的 `Basic.Properties` 数据结构中。最后，消息内容被编组到一定数量的消息体帧中。

注意　虽然帧的默认大小为 131 KB，但客户端库在连接过程中可以协商更大或更小的帧大小上限，帧中的字节数最大可以达到一个 32 位值。

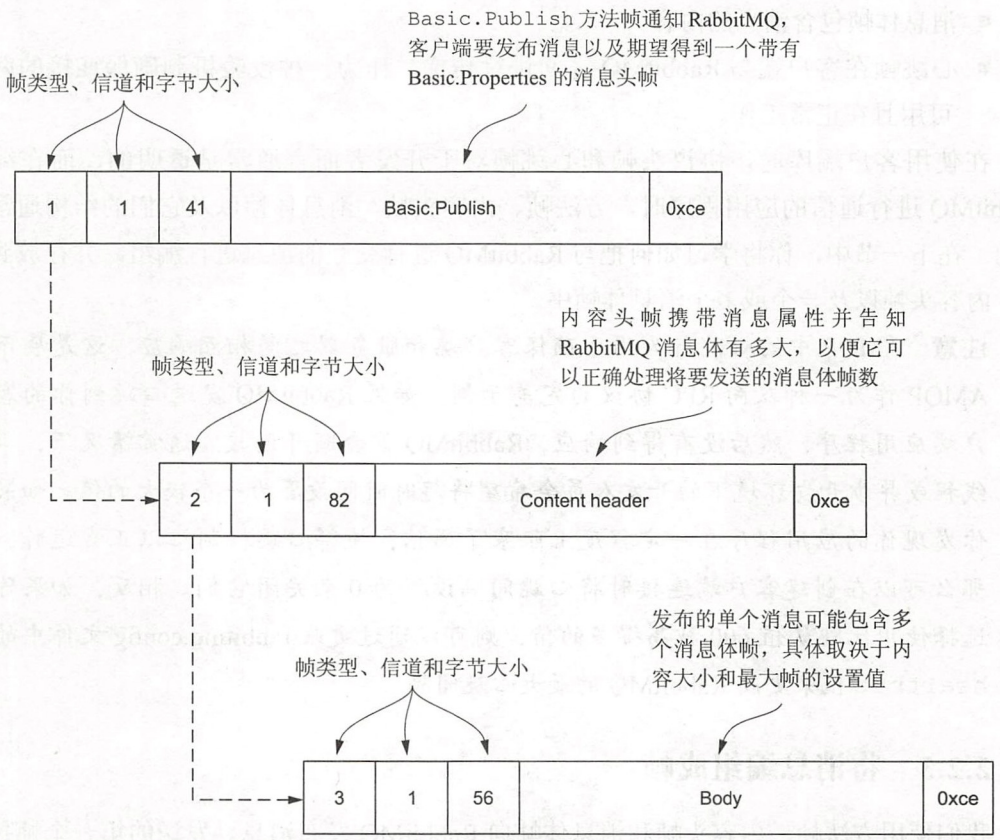


图 2.4　发布到 RabbitMQ 中的单个消息由三种帧类型组成：供 Basic.Publish RPC 调用的方法帧、消息头帧，以及一个或多个消息体帧

为了更高效地处理并最小化传输的数据大小，方法帧和内容头帧中的内容是人眼不可读的二进制打包数据。而与方法帧和内容头帧不同，在消息体帧内部携带的消息内容没有进行任何打包或编码，可以包含从纯文本到二进制图像数据的任何内容。

为了进一步展示 AMQP 消息的内部构造，我们来对这三种帧类型做进一步讨论。

2.2.4　方法帧结构

方法帧携带着构建 RPC 请求所需的类、方法以及相关参数。在图 2.5 中，携带 Basic.Publish 命令的方法帧中包含着描述该命令的二进制打包数据以及与其一起传递的请求参数。前两个字段是 Basic 类和 Publish 方法的数字表示。这些字段后面跟着交换器和路

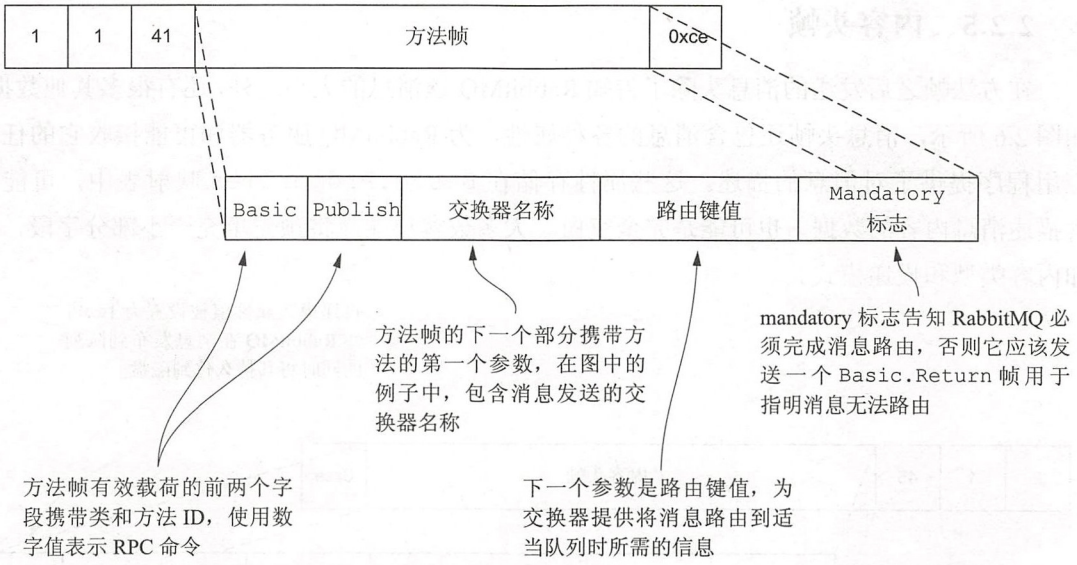


图 2.5　Basic.Publish 方法帧由五个组件组成：用于标识 Basic.Publish RPC 请求的类和方法类型、交换器名称、路由键值和 mandatory 标志

由键的字符串值。正如前文所述，这些属性告知 RabbitMQ 如何路由消息。Mandatory 标志则告知 RabbitMQ 消息必须投递成功，否则发布消息的过程就应该是失败的。

方法帧有效载荷中的每个数据值都是按照数据类型采用特定的格式进行编码。这种编码格式旨在最大限度地减少网络传输中的字节大小，提供数据完整性，并确保数据编组和解组速度尽可能快。实际的格式根据数据类型的不同而不同，但通常表现为一个字节后跟数字数据，或者是一个字节后跟一个字节大小的字段，其后则是文本数据。

注意　通常情况下，使用 Basic.Publish RPC 请求发送消息是一个单向会话。事实上，AMQP 规范提供了一种通用规则，即成功通常是无声的，而错误应该是尽可能嘈杂和干扰的。但是，如果你在发布消息时使用了 mandatory 标志，则应用程序应该监听从 RabbitMQ 发送回来的 Basic.Return 命令。如果 RabbitMQ 不能满足 mandatory 标志设置的要求，它将在同一个信道上发送一个 Basic.Return 命令到你的客户端。有关 Basic.Return 的更多信息可参阅第 4 章。

2.2.5 内容头帧

在方法帧之后发送的消息头除了告知 RabbitMQ 该消息的大小之外，还有很多其他数据。如图 2.6 所示，消息头帧还包含消息的各种属性，为 RabbitMQ 服务器和可能接收它的任何应用程序提供了对消息的描述。这些属性存储在 `Basic.Properties` 映射表中，可能包含描述消息内容的数据，也可能是完全空白。大多数客户端库将预先填充一小部分字段，比如内容类型和投递模式。

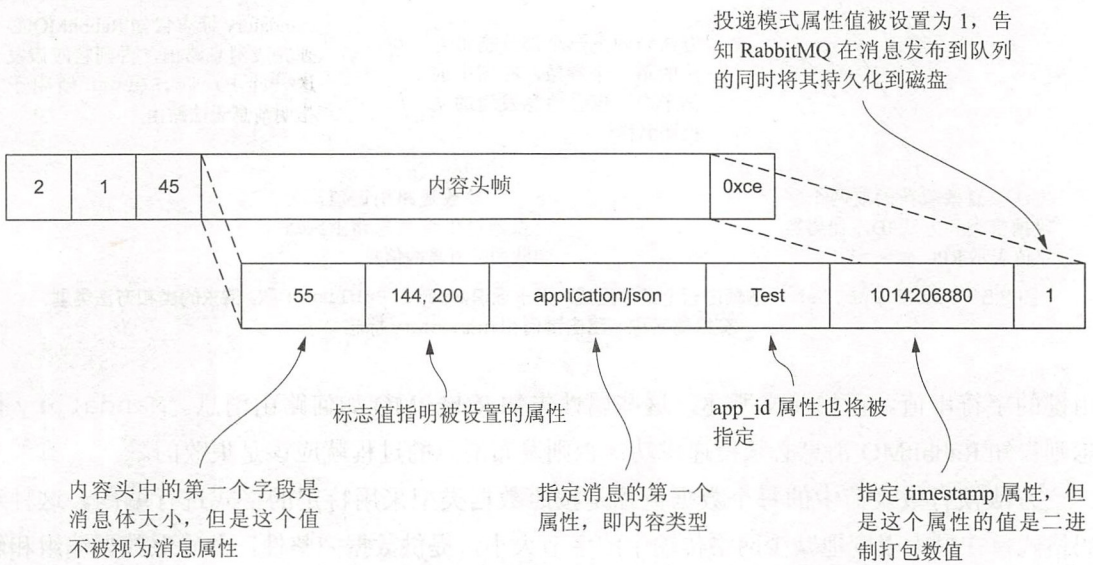


图 2.6 消息头携带着消息体大小以及 Basic.Properties 属性表

属性是编写消息的强大工具。它们可以用来在发布者和消费者之间就消息的内容创建契约，从而允许对消息进行大量的定制操作。在第 3 章中，你将学习 `Basic.Properties` 以及各个字段所携带数据结构的多种潜在用法。

2.2.6 消息体帧

消息的消息体帧与正在传输的数据类型无关，并且可能包含二进制或文本数据。无论你是发送如 JPEG 图片的二进制数据，或是序列化之后的 JSON、XML 格式数据，消息体帧都是消息中包含实际消息数据的结构（见图 2.7）。

消息属性和消息体组合在一起构成了数据的强大封装格式。将消息的描述性属性与内容无关的消息体结合起来，确保你可以使用 RabbitMQ 来处理你认为合适的任何类型的数据。

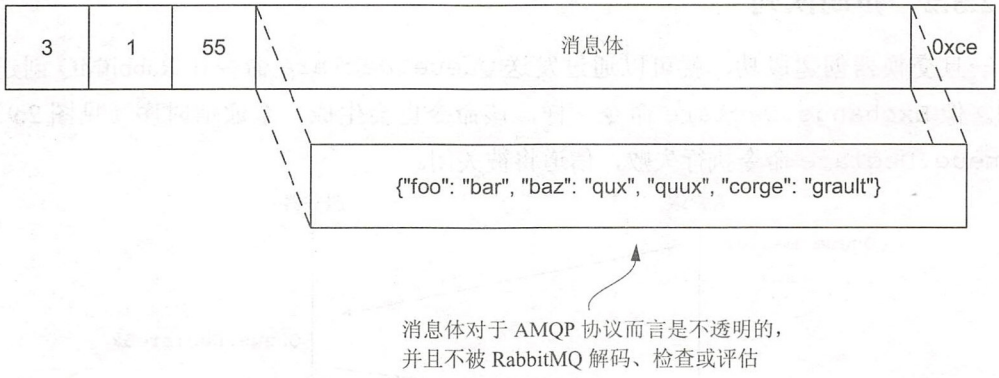


图 2.7　嵌入在 AMQP 帧中的消息体

2.3　使用协议

在将消息发布到队列之前，有几个与配置相关的步骤需要注意。至少，你必须设置交换器和队列，然后将它们绑定在一起。

但在实际执行这些步骤之前，我们来看一下在协议级别需要发生什么事情以使消息能够发布、路由、存到队列以及投递，首当其冲的是设置用于提供消息路由的交换器。

2.3.1　声明交换器

和队列一样，交换器在 AMQ 模型中是一等公民。因此，它们在 AMQP 规范中都有自己的类。使用 `Exchange.Declare` 命令可以创建交换器，该命令提供了定义交换器名称和类型的参数，以及用于消息处理的其他元数据。

一旦命令被发送，RabbitMQ 在创建了交换器之后将发送一个 `Exchange.DeclareOk` 方法帧作为响应（见图 2.8）。如果出于某种原因命令执行失败，则 RabbitMQ 将使用 `Channel.Close` 命令关闭发送 `Channel.Declare` 命令的信道。该响应将包含一个数字回复编码和文本值，用于说明 `Exchange.Declare` 失败并关闭信道的原因。

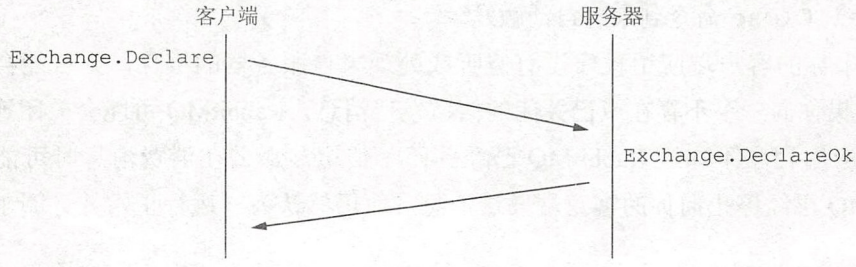


图 2.8　声明交换器时的通信时序

2.3.2 声明队列

一旦交换器创建成功，就可以通过发送 Queue.Declare 命令让 RabbitMQ 创建一个队列。像 Exchange.Declare 命令一样，该命令也会生成一个通信时序（见图 2.9），如果 Queue.Declare 命令执行失败，信道将被关闭。

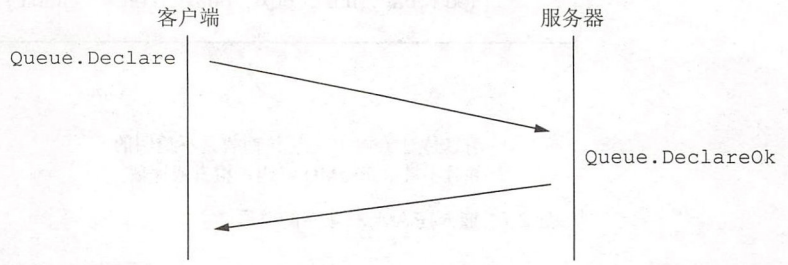


图 2.9 声明队列时的时序图，包括 Queue.Declare 命令以及 Queue.DeclareOk 响应

在声明一个队列时，多次发送同一个 Queue.Declare 命令并不会有任何副作用。RabbitMQ 不会处理后续的队列声明，只会返回队列相关的有用信息，比如队列中待处理消息的数量以及订阅该队列的消费者数量。

优雅地处理错误

当你尝试声明一个与现有队列同名的新队列时，如果新队列的属性与现有队列不一样，那么 RabbitMQ 将关闭发出 RPC 请求的信道。这种行为与你的客户端应用程序向代理服务器发送命令时可能发生的任何其他类型的错误一致。例如，如果一个用户发出 Queue.Declare 命令，而该用户并没有在虚拟主机上被配置相应的访问权限时，该信道将关闭并显示 403 错误。

要正确处理错误，你的客户端应用程序应该监听来自 RabbitMQ 的 Channel.Close 命令以便能够正确响应。某些客户端库可能会将此信息当作一种异常，然后让你的应用程序去处理。而其他客户端可能会使用回调风格，通过注册一个回调方法在 Channel.Close 命令到来时自动触发。

如果你的客户端应用程序没有监听或处理来自服务器的事件，则可能会丢失消息。 如果你向一个不存在或已关闭的信道发送消息，RabbitMQ 可能会关闭连接。如果你的应用程序不知道 RabbitMQ 已经关闭了信道，那么在消费消息时可能不知道 RabbitMQ 已经停止向你的客户端发送消息，而仍然认为它运行正常并订阅了一个空队列。

2.3.3　绑定队列到交换器

一旦创建了交换器和队列，是时候将它们绑定在一起了。如同 Queue.Declare 命令，将队列绑定到交换器的 Queue.Bind 命令每次只能指定一个队列。与 Exchange.Declare 和 Queue.Declare 命令类似，在发出 Queue.Bind 命令后，如果处理成功，那么你的应用程序会收到一个 Queue.BindOk 方法帧（见图 2.10）。

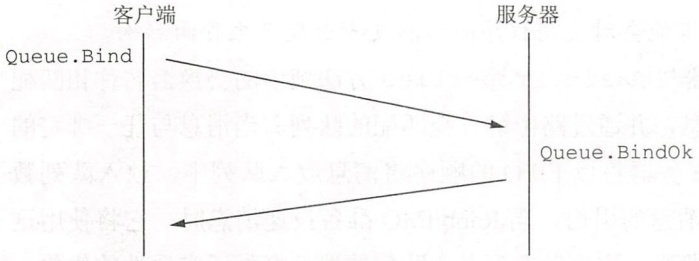


图 2.10　客户端成功发出 Queue.Bind 命令之后通过一个路由键将队列绑定到交换器，客户端将收到一个 Queue.BindOk 方法帧作为响应

作为 RabbitMQ 服务器和客户端之间的 RPC 交互基本示例，Exchange.Declare、Queue.Declare 和 Queue.Bind 命令展示了 AMQP 规范中所有同步命令类似的通用模式。但是有一些异步命令的执行效果与这种“Action”和“ActionOk”的简单模式有所不同。这些命令处理与 RabbitMQ 之间的消息发送和接收。

2.3.4　发布消息到 RabbitMQ

正如前面所介绍的，当发布消息到 RabbitMQ 时，多个帧封装了发送到服务器的消息数据。在实际的消息内容到达 RabbitMQ 之前，客户端应用程序发送一个 Basic.Publish 方法帧、一个内容头帧和至少一个消息体帧（见图 2.11）。

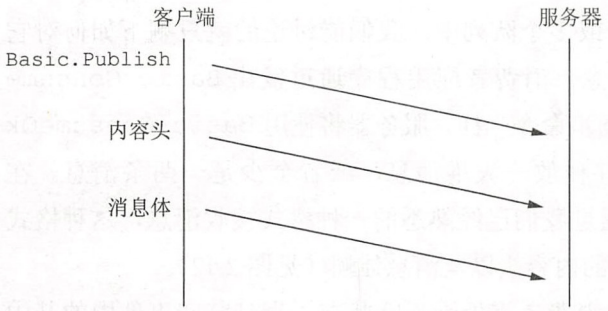


图 2.11　当发送消息到 RabbitMQ 时，至少需要发送三个帧：Basic.Publish 方法帧、内容头帧和消息体帧

当 RabbitMQ 接收到一个消息的所有帧并确定下一步操作之前，它将检查方法帧以获取它所需要的信息。Basic.Publish 方法帧携带消息的交换器名称和路由键。在评估这些数据时，RabbitMQ 会尝试将 Basic.Publish 帧中的交换器名称与配置交换器的数据库进行匹配。

提示 默认情况下，如果你使用 RabbitMQ 配置中不存在的交换器发布消息，它将自动丢弃该消息。要想确保你的消息成功投递，请在发布时将 mandatory 标志设置为 true，或者使用投递确认机制。这些选项会在第 4 章中详细介绍。请注意，使用这些方法中的任何一种都可能会对应用程序的消息发布速度产生负面影响。

当 RabbitMQ 发现某一个交换器与 Basic.Properties 方法帧中的交换器名称相匹配时，它将判断该交换器中的绑定信息，并通过路由键寻找匹配的队列。当消息与任一绑定的队列符合匹配标准时，RabbitMQ 服务器将以 FIFO 的顺序将消息放入队列中。放入队列数据结构中的并不是实际消息，而是消息的引用。当 RabbitMQ 准备投递消息时，它将使用这个引用来编组消息并通过网络进行发送。这为发布到多个队列的消息提供了实质性的优化。当把消息发送到多个目标时，只保存消息的一个实例会占用较少的物理内存。某一个队列中的消息处理方式，无论是被消费、过期还是等待消费，都不会影响该消息在其他队列中的处理方式。一旦 RabbitMQ 不再需要这个消息，因为它的所有副本都已经被投递或者被删除了，单个消息数据将被从 RabbitMQ 的内存中移除。

默认情况下，只要没有消费者正在监听队列，消息就会被存储在队列中。当添加更多消息时，队列的大小也会随之增加。RabbitMQ 可以将这些消息保存在内存中或写入磁盘，具体取决于消息 Basic.Properties 中指定的 delivery-mode 属性。delivery-mode 属性非常重要，我们将在第 3 章以及第 4 章中做详细介绍。

2.3.5 从 RabbitMQ 中消费消息

一旦发布的消息被路由并保存到一个或多个队列中，我们能讨论的就只剩下如何对它进行消费。要消费 RabbitMQ 队列中的消息，消费者应用程序通过发出 Basic.Consume 命令来订阅 RabbitMQ 中的队列。像其他同步命令一样，服务器将使用 Basic.ConsumeOk 进行响应，让客户端知道它将打开闸门并释放一大堆消息，或者至少是一两条消息。在 RabbitMQ 的说明文档中，消费者将开始通过我们已经熟悉的一种格式接收消息，这种格式包括对应的 Basic.Deliver 方法和它们的内容头以及消息体帧（见图 2.12）。

一旦发送完 Basic.Consume 命令，消费者将处于活跃状态，直到某些事件中的其中一个被触发。如果消费者想要停止接收消息，则可以发出一个 Basic.Cancel 命令。值得

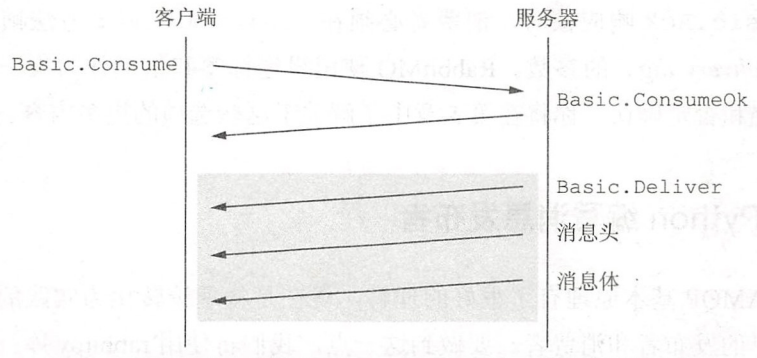


图 2.12　订阅队列和接收消息时客户端和服务器之间的逻辑帧投递顺序

注意的是，这个命令是异步发出的，而 RabbitMQ 可能仍然在发送消息，所以消费者在接收到一个 Basic.CancelOk 响应帧之前仍然可以接收到 RabbitMQ 预分配给它的任意数量的消息。

消费消息时，有几个设置可以让 RabbitMQ 知道你想如何接收它们。其中的一个设置是用 Basic.Consume 命令中的 no_ack 参数。当设置该参数为 true 时，RabbitMQ 将连续发送消息直到消费者发送一个 Basic.Cancel 命令或消费者断开连接为止。如果 no_ack 标志被设置为 false，则消费者必须通过发送 Basic.Ack RPC 请求来确认收到的每条消息（见图 2.13）。

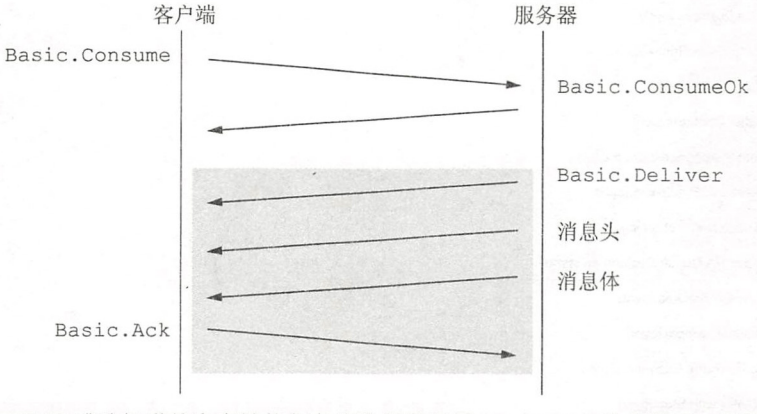


图 2.13　RabbitMQ 成功投递给客户端的每条消息都将通过 Basic.Ack 进行响应，直到一个 Basic.Cancel 命令被发送为止。如果设置了 no_ack，则忽略 Basic.Ack 步骤

当发送 Basic.Ack 响应帧时，消费者必须在 Basic.Deliver 方法帧中传递一个名为投递标签（*delivery tag*）的参数。RabbitMQ 使用投递标签和信道作为唯一标识符来实现消息确认、拒绝和否定确认。你将在第 5 章中了解关于这些选项的更多内容。

2.4 用 Python 编写消息发布者

你已经对 AMQP 基本原理有了很好的理解，现在是将理论转化为实践的时候了，我们将同时编写消息的发布者和消费者。要做到这一点，我们将使用 rabbitpy 库。有许多库可以与 RabbitMQ 进行通信，但是我编写了 rabbitpy 作为本书的辅助工具，在提供清晰的 AMQP 命令语法时确保编程示例的简洁性和正确性。如果你还没有安装 rabbitpy，请按照附录中的 VM 安装说明进行安装。

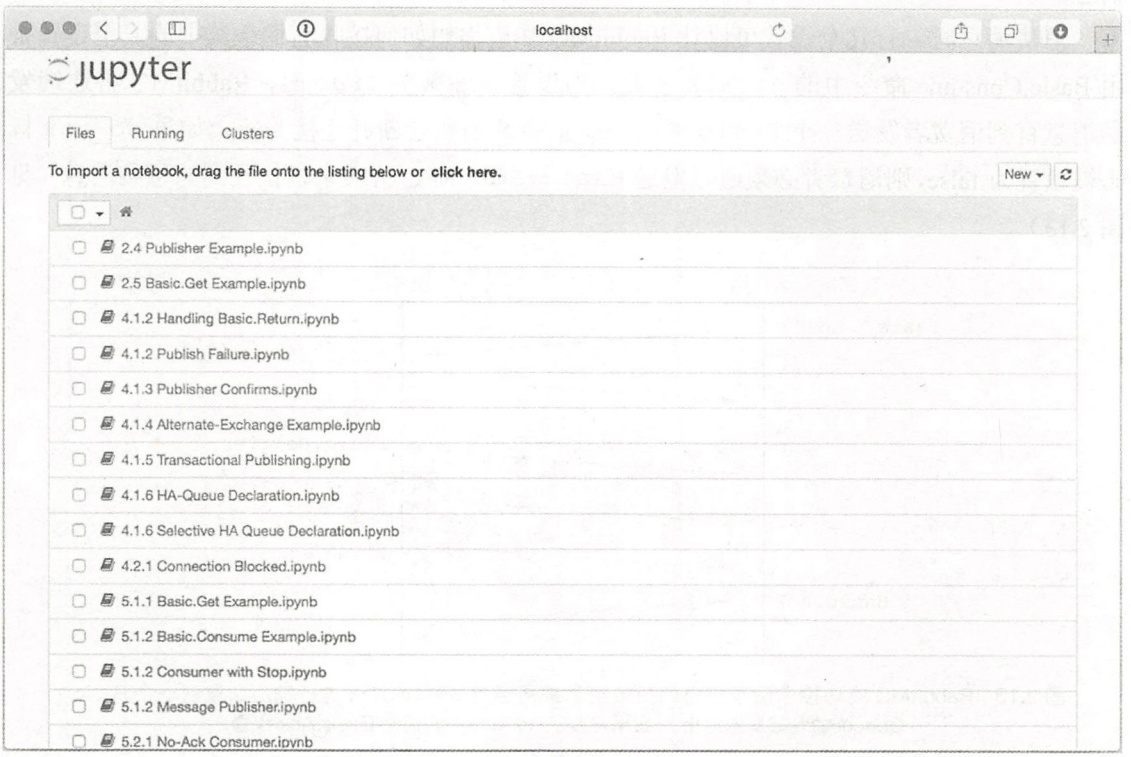


图 2.14　IPython 笔记文件索引界面

要开始这个练习，你将使用安装在 RabbitMQ in Depth 虚拟机中的 IPython 笔记文件（Notebook）服务器，该服务器是 RabbitMQ 的组成部分。如果你还没有进行安装，请按照附录中列出的步骤在本地计算机上设置虚拟机。打开浏览器访问 http://localhost:8888，你会看到类似于图 2.14 的页面。

索引中的“2.4 发布者示例”笔记文件包含此页面中列出的所有代码，这些代码用于和 RabbitMQ 进行通信。你必须导入 rabbitpy 库以便 Python 解释器能够使用它：

```
In [1]: # Import the RabbitMQ client Library
        import rabbitpy
```

点击工具栏上的“播放”按钮或“运行单元”按钮，或者按下 Shift-Enter 键，将会执行该单元中所包含的代码。在笔记文件的第一个单元中，rabbitpy 库将被导入。

你还应该看到星号（*）变成了数字 1。活动单元格从第一个自动进入下一个。在阅读这个示例代码时，应该执行你所看到的每个单元，从而不断向前学习 IPython 笔记文件中的代码。

现在，在导入 rabbitpy 库后，需要创建一个 AMQP 连接 URL。URL 的格式与 HTTP 请求的格式非常相似：

```
In [2]: # Speciry the  URL  to connect to
        url = 'amqp://quest:quest@localhost:5672/%2F'
```

该 AMQP URL 指定你将使用用户名“guest”和密码“guest”对一个标准 AMQP 连接进行连接。它将使你连接到端口号为 5672 的本地机器，该机器位于默认的“/”虚拟主机中。通过这个 URL，你将使用默认配置连接到本地机器上的 RabbitMQ。如果你在远程服务器上搭建了 RabbitMQ，或者更改了 RabbitMQ 代理服务器的配置，则必须相应地更改这些值。

现在我们已经设置了 URL，接下去需要打开通向 RabbitMQ 的连接：

```
In [3]: # Connect to RabbitMQ USing the URL above
        connection = rabbitpy.Connection(url)
```

如果你没有收到异常，那么现在已经连接到了 RabbitMQ。如果收到了一个异常，最可能的情况是 RabbitMQ 没有运行在你的本地机器上。请确保它正在运行之后进行重试。

如果连接已经成功，接下去就是打开一个信道，然后与 RabbitMQ 进行通信：

```
In [4]: # Open a new channel on the connection
        channel = connection.channel()
```

在信道打开的情况下，现在可以通过创建 rabbitpy.Exchange 类的新实例来声明交换器。传入信道以及你想要创建的交换器名称。我建议现在使用 chapter2-example 作为交换器的名称。

```
In [5]: # Create a new exchange object, passing in the chanel to use
        exchange = rabbitpy.Exchange(channel,'chapter2-example')
```

一旦构建完成，使用交换器对象中的 declare 方法发送命令，并在 RabbitMQ 中声明交换器：

```
In [6]: # Declare the exchange on the RabbitMQ server
        exchange.declare()
```

现在你已经声明了交换器，就可以创建队列并将其绑定到交换器。为此，首先创建 Queue 对象，并传入信道和队列的名称。在下面的例子中，队列的名字就是 example。

```
In [7]: # Create a new exchange object, passing in the chanel to use
        queue = rabbitpy.Queue(channel,'example')
```

一旦创建了对象并返回名为 queue 的队列实例，就可以使用 declare 方法将 Queue.Declare 命令发送到 RabbitMQ。你应该看到一个具有 Python 元组数据结构的输出行，其中包含队列中的消息数量和队列的消费者数量。元组（tuple）是一个 Python 对象的不可变集合。 在这个例子中，它们都是整数值。

```
In [8]: # Declare the  queue on the RabbitMQ server
        queue.declare()
Out[8]: (10, 0)
```

现在队列已经被创建了，你必须绑定它才能接收消息。要将队列绑定到交换器，可以调用队列对象的 bind 方法，传入交换器和路由键参数来发送 Queue.Bind 命令。在以下示例中，路由键是 example-routing-key。当这个单元执行返回时，你应该看到输出 True，表示绑定成功。

```
In [9]: # Bind the  queue to the exchange on the RabbitMQ server
        queue.bind(exchange,'example-routing-key')
Out[9]: True
```

在你的应用程序中，我建议你使用合适的语义并以句点分隔的关键词来命名你的路由键。《Python 之禅（*Zen of Python*）》一书有云："命名空间是一个好主意—— 让我们尽量使用它！"。在 RabbitMQ 中同样如此。通过使用句点分隔的关键词，你将能够根据路由键的模式和子段来路由消息。关于这点将在第 6 章中进一步讨论。

提示　*队列和交换器名称以及路由键可以包括 Unicode 字符。*

通过创建和绑定交换器和队列，你现在可以将测试消息发布到 RabbitMQ 并存储在 `example` 队列中。为了确保有足够的消息可供使用，以下示例将 10 条测试消息发布到队列中。

```
In [10]: # fossage_number in range(0, 10):
        message = rabbitpy.Message(channel,
                                   'Test messxge #%i' % message number,
                                   {'content_type': 'text/plain'},
                                   opinionated=True)
        message.publish(exchange, 'example-routing-key')
```

为了发布测试消息，在每个循环中创建一个新的 `rabbitpy.Message` 对象，并传入信道、消息体和消息属性字典。一旦消息创建成功，`publish` 方法就会被调用，该方法创建 `Basic.Publish` 帧、内容头帧和一个消息体帧，并将它们全部投递给 RabbitMQ。

提示　*当你为生产环境编写发布者应用程序时，请使用 JSON 或 XML 等数据序列化格式以便消费者可以轻松地反序列化消息，这样在解决可能出现的任何问题时更易于阅读。*

你现在应该进入 RabbitMQ Web 管理控制台，看看你的消息是否已经进入队列：打开你的 Web 浏览器并访问 http://localhost:15672/ # /queues/% 2F/example 中的管理界面（如果你的代理服务器位于不同的计算机上，请将 URL 中的 *localhost* 更改为相应的服务器）。一旦通过身份验证，应该看到类似于图 2.15 所示的屏幕截图页面。

在该页的页面底部你会看到"获取消息（Get Messages）"模块。如果将"消息"字段值从 1 依次递增到 10，然后单击"获取消息"，则应该看到之前发布的 10 条消息中的每一条。确保将"重发队列（Requeue）"字段值设置为"Yes"。该字段告诉 RabbitMQ，在获取消息以便在管理界面中进行显示的同时，将它们添加回队列中。如果你没有做这一步，没关系；回去重新运行代码发布。

图 2.15　RabbitMQ Web 管理界面展示了队列中按顺序进行处理的 10 条消息

2.5　从 RabbitMQ 中获取消息

现在你已经知道如何发布消息，接下来讨论如何获取它们。以下代码清单将上一节讨论的发布者代码中可复用的连接元素汇集在一起，从而允许你从 RabbitMQ 获取消息。此代码位于“2.5 Basic.Get 示例”笔记文件中。使用 IPython 笔记文件界面时，该笔记文件有六个单元格。你可以单击“单元”下拉菜单，然后执行“全部运行”，而不是像上例中那样每次运行一个单元。

```
import rabbitpy

url = 'amqp://guest:guest@localhost:5672/%2F'
connection = rabbitpy.Connection(url)
channel = connection.channel()
queue = rabbitpy.Queue(channel, 'example')


while len(queue) > 0:
  message = queue.get()
  print 'Message:'
```

创建一个新的连接对象，连接到 RabbitMQ

打开一个信道进行通信

创建一个新的队列对象来获取消息

在队列有消息时执行循环

检索消息

```
print ' ID: %s' % message.properties['message_id']
print ' Time: %s' % message.properties['timestamp'].isoformat()
print ' Body: %s' % message.body
message.ack()
```

从队列中获取消息

打印 timestamp 属性，格式为 ISO 8601 时间戳

打印消息体

RabbitMQ 确认收到消息

输入并执行前面的消费者代码，应该看到之前发布的 10 条消息中的每一条。如果仔细观察，你可能已经注意到，尽管在发布消息时没有指定 `message_id` 或 `timestamp` 属性，但从消费者打印出来的每条消息中都有它们。如果不指定它们，rabbitpy 客户端库将自动为你填充这些属性。另外，如果你发送了一个 Python `dict` 结构作为消息，rabbitpy 会自动将数据序列化为 JSON 格式，并将 content-type 属性设置为 `application/json`。

2.6　小结

AMQP 0.9.1 规范定义了一种在 RabbitMQ 服务器和客户端之间进行通信的通信协议，该协议使用 RPC 风格的命令。现在你已经知道这些命令的结构以及协议的具体功能，并能够更好地编写与 RabbitMQ 交互的应用程序以及处理相关问题。你已经了解了大部分与 RabbitMQ 进行通信以实现消息发布和消费的过程。相较你已经实现的基于 RabbitMQ 的这些示例程序，许多应用程序并不会包含比它们更多的代码。

在下一章中你将学习更多关于使用消息属性的知识，以便你的发布者和消费者使用应用程序交换消息的共同契约。

第 3 章　消息属性详解

本章概要：

- 消息属性以及对消息投递的影响
- 使用消息属性在发布者和消费者之间创建契约

在第 1 章中，我详细介绍了如何解耦会员登录事件与数据库写入操作，以解决会员登录网站延迟的问题。很快，我们整个工程团队就明白这样做的好处，使用松耦合的数据库写入架构自然而然就诞生了。随着时间的推移，我们开始在新开发的应用中使用这种架构。我们不再只是处理会员登录事件，而是使用这种架构处理账户删除、电子邮件生成以及任何可以异步执行的应用程序事件。事件通过消息总线发布到消费者应用程序，每个事件都执行自己特有的任务。起初，我们对信息包含的内容以及表现形式做了一些简单的设想，但很快就明白需要进行标准化。

由于存在不同的消息类型以及缺少消息格式的标准化，我们难以预测特定的消息类型如何被序列化以及包含哪些具体数据。开发人员会使用一种对自身应用程序有意义的格式

发布消息，而这种格式只适合这个应用系统。虽然他们完成了自己的任务，但是这个想法是短视的。我们开始观察到，消息可以在多个应用程序中重复使用，这样随意的消息格式就变成了一个问题。这些问题和其他相关问题所带来的痛苦日益增长，为了缓解这些痛苦，我们更加注意描述正在发送的消息，无论是在文档中还是作为消息本身的一部分。

为了提供消息自描述的一致方法，我们查看了 AMQP 的 Basic.Properties 数据结构，通过 AMQP 发布到 RabbitMQ 的每条消息中都包含这一结构。利用 Basic.Properties 提供了一种途径，使得消费者可以更加智能——消费者应用程序可以自动反序列化消息，在处理消息之前验证消息的来源及其类型等。在本章中，我们将深入研究 Basic.Properties，涵盖其中的每个属性及其潜在用法。

3.1　合理使用属性

回想一下第 2 章中的内容，当你使用 RabbitMQ 发布消息时，消息由 AMQP 规范中的三个低层帧类型组成：Basic.Publish 方法帧、内容头帧和消息体帧。这三种帧类型按顺序一起工作，以便将消息传递到它应该去的地方并确保它们到达时完好无损（见图 3.1）。

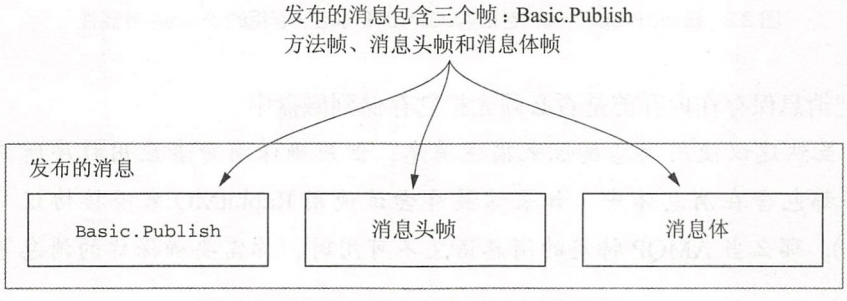


图 3.1　发布到 RabbitMQ 中消息的三个组件

包含在消息头帧中的消息属性是一组预定义的值，这些值通过 Basic.Properties 数据结构进行指定（见图 3.2）。某些属性（如 delivery-mode）在 AMQP 规范中具有明确的含义，而有些属性（如 type）则没有明确的规范。

在某些情况下，RabbitMQ 使用明确定义的属性来实现消息的特定行为。前面提到的 delivery-mode 属性就是一个例子。在将消息放入队列时，delivery-mode 值将告诉

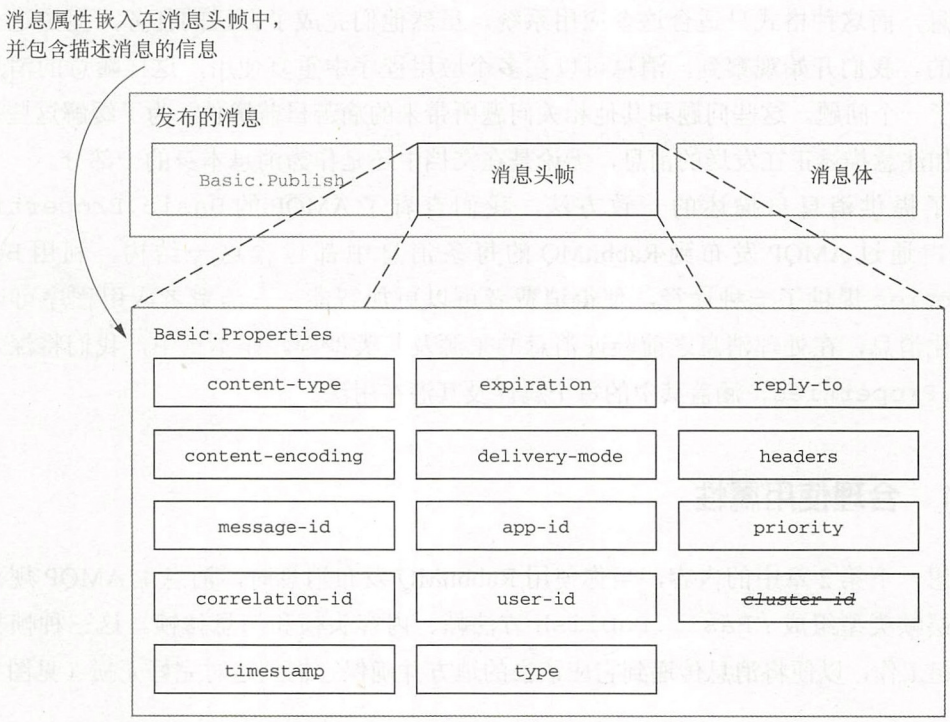


图 3.2　Basic.Properties，包括 AMQP-0-8 以后已弃用的 cluster-id 属性

RabbitMQ 把消息保存在内存前是否必须先把它存储到磁盘中。

提示　虽然建议使用消息属性来描述消息，但应确保消费者应用程序所需的所有数据都包含在消息体中。如果你最终尝试使用 RabbitMQ 来桥接协议（例如 MQTT），那么当 AMQP 特定的消息语义不可用时，你需要确保你的消息不会丢失含义。

在消息标准化的发展过程中，AMQP 消息属性为定义和传输消息元数据提供了一个有用的起点。这些元数据反过来使读者能够在消息发布者和消费者之间建立严格的契约。包含 content-type、消息类型（type）、timestamp 和应用程序标识（app-id）在内的很多属性都已经被证明是非常有用的，它们不仅可用于在过程中实现数据一致性，也适合于日常操作。简而言之，通过使用消息属性，你可以创建自描述消息，类似于将 XML 视为一种自描述数据标记语言。

在本章中，我们将分析图 3.2 中所列举的各个基本属性：

- 使用 content-type 属性让消费者知道如何解释消息体。
- 使用 content-encoding 属性来指示消息体使用某种特殊的方式进行压缩或编码。
- 填充 message-id 和 correlation-id 来唯一标识消息和消息响应，用于在工作流程中实现消息跟踪。
- 利用 timestamp 属性减少消息大小，并创建一个规范定义来描述消息创建时间。
- 使用 expiration 属性表明消息过期。
- 告诉 RabbitMQ 使用 delivery-mode 将消息写入磁盘或内存队列。
- 使用 app-id 和 user-id 来帮助追踪出现问题的消息发布者应用程序。
- 使用 type 属性来定义发布者和消费者之间的契约。
- 使用 reply-to 属性实现响应消息的路由。
- 使用 headers 映射表定义自由格式的属性和实现 RabbitMQ 路由。

我们还会涉及为什么要避免使用 priority 属性，并介绍 cluster-id 属性背后发生的事情以及为什么不能使用它。

我将按照这个列表的顺序讨论这些属性，但是我还会在本章末尾给出一个简表，按照字母顺序列出每个属性及其数据类型，并使用一个标识符来表明它可用于代理服务器还是应用程序，最后给出使用说明。

注意　当我使用“契约（contract）”这个术语来描述消息通信时，我指的是一种确定消息格式和内容的规范。在编程中，这个术语经常用来描述 API、对象和系统的预定义规范。契约规范通常包含有关发送和接收数据的精确信息，例如数据类型、格式以及各种需要遵守的条件。

3.2　使用 content-type 属性创建显式的消息契约

正如我很快发现的那样，对于通过 RabbitMQ 发布的消息，我们很容易找到新的用途。我们最初的消费者应用程序是用 Python 编写的，但不久之后，使用 PHP、Java 和 C 语言编写的应用程序同样成为了消息的消费者。

当消息的有效载荷格式无法自描述时，你的应用程序会倾向使用一种隐式契约，这种隐式契约天生容易出错，所以你的应用程序非常可能出现问题。通过使用自描述消息，程序员和消费者应用程序不需要猜测如何反序列化消息中的数据，甚至根本不需要执行反序列化操作。

Basic.Properties 数据结构包含了 content-type 属性，该属性用于传输消息体中的数据格式（见图 3.3）。

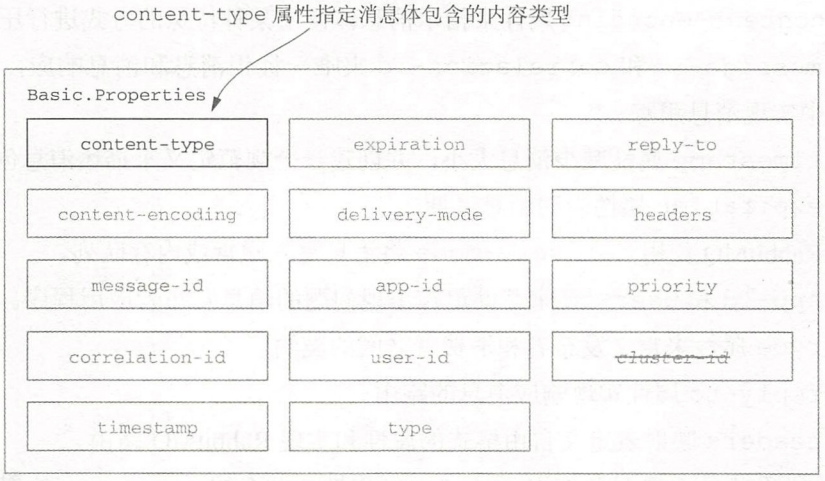


图 3.3 content-type 属性是 Basic.Properties 中的第一个属性

如同各种标准化的 HTTP 规范，content-type 传输消息体的 MIME 类型。例如，如果你的应用程序正在发送 JSON 序列化的数据值，那么将 content-type 属性设置为 application/json 将允许尚待开发的消费者应用程序在收到消息时检查消息类型并对消息进行正确解码。

关于自描述消息和消息内容的思考

使用标准的序列化格式如 JSON、Msgpack（http://msgpack.org/）或 XML 是明智的。这些格式允许使用任何编程语言编写任意数量的消费者应用程序。因为数据是以这些格式自我描述的，所以编写潜在的消费者应用程序会很容易，并且在核心应用程序之外的网络环境上对消息解码也会变得更加容易。

另外，通过 content-type 属性指定序列化格式，可以更好地支持适应未来的消费者应用程序。当消费者可以自动识别它们所支持的序列化格式，并且可以选择性地处理消息时，不必担心在使用新的序列化格式并将其路由到相同的队列时会发生什么状况。

如果你在消费者代码中使用一个框架，那么你可能需要明白它如何处理收到的消息。在消费者代码处理消息之前，通过框架对其进行预处理，消息体可以自动地被反序列化

并加载到你所使用编程语言的本地数据结构中。例如，在 Python 中，你的框架可以从 content-type 消息头中检测到消息序列化类型，并使用该类型自动反序列化消息体并将内容放入一个 dict、list 或其他原生数据类型中。这最终将降低消费者应用程序中代码的复杂性。

3.3　通过 gzip 和 content-encoding 属性压缩消息大小

默认情况下，通过 AMQP 发送的消息并不会被压缩。在处理如 XML 这种过于繁杂的标记语言时，甚至在消息数量较大的场景下处理像 JSON 或 YAML 等较少使用标记的轻量级格式时，这都可能会是个问题。你的发布者可以在发布消息之前压缩消息，并在收到消息时进行解压缩，如同我们使用 gzip 在服务器上压缩网页然后在浏览器端实时解压缩这些网页之后再进行展示一样。

为了显式指定这个过程，AMQP 提供了 content-encoding 属性（见图 3.4）。

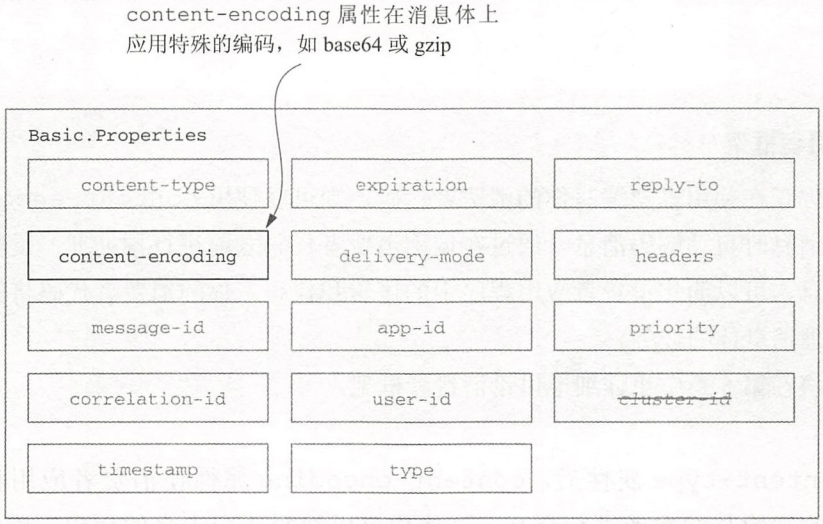


图 3.4　content-encoding 属性指定是否在消息体上使用了特殊的编码方式

最好不要更改生产环境中已发布和已消费消息的契约，从而最大限度地减少对现有代码的潜在影响。但是，如果消息大小影响整体性能和稳定性，那么使用 content-encoding

消息头将允许消费者对消息进行预判断，确保它们可以解码消息体中发送的任何格式。

注意 不要混淆 content-encoding 和 content-type。与 HTTP 规范一样，content-encoding 用于指示 content-type 之外的某种编码级别。它是一个修饰符字段，通常用于表明消息体的内容已经使用 gzip 或其他形式的压缩方式进行了压缩。某些 AMQP 客户端自动将 content-encoding 值设置为 UTF-8，但这是不正确的行为。AMQP 规范规定 content-encoding 用于存储 MIME 内容编码。

我们做一个对比，MIME 电子邮件标记使用 content-encoding 字段来指定电子邮件每个不同部分的编码方式。在电子邮件中，最常见的编码类型是 Base64 和 QuotedPrintable。Base64 编码用于确保消息中传输的二进制数据不违反纯文本的 SMTP 协议。例如，如果你正在创建基于 HTML 的电子邮件，该电子邮件内嵌了图像，那么该嵌入的图像可能就是通过 Base64 进行了编码。

但是，与 SMTP 不同，AMQP 是一种二进制协议。消息体中的内容以原生的状态传输，不在消息编组和解组的过程中进行编码或转换。任何格式的内容都可以进行传输而不用担心违反协议。

应用消费者框架

如果你正在使用框架编写你的消费者代码，则可以使用 content-encoding 属性在收到消息时自动解码消息。通过在调用消费者代码之前进行预处理、反序列化和解压缩消息，可以简化消费者应用程序中的逻辑和代码。你的消费者代码将能够专注于那些处理消息体的任务。

我们将在第 5 章中更详细地讨论消费者框架。

结合 content-type 属性后，content-encoding 属性使消费者应用程序能够基于一种明确的契约与发布者进行交互。这使你可以编写适应未来的代码，确保你的代码能够抵御由于消息格式变更导致的意外错误。例如，在应用程序的生命周期中，你可能会发现 bzip2 压缩更加适合你的消息内容。如果你编写消费者应用程序来检查 content-encoding 属性，则可以拒绝那些不能解码的消息。只知道如何使用 zlib 或 deflate 进行解压缩的消费者便会拒绝新的 bzip2 压缩消息，并把它们留在队列中供其他消费者应用程序去解压缩 bzip2 消息。

3.4　使用 message-id 和 correlation-id 引用消息

在 AMQP 规范中，message-id 和 correlation-id 是“应用级别使用”的属性，并没有提供正式的行为定义（见图 3.5）。这意味着就规范而言，你可以利用它们实现任何目的。这两个字段允许多达 255 个字节的 UTF-8 编码数据，并以未压缩的方式存储在 Basic.Properties 数据结构中。

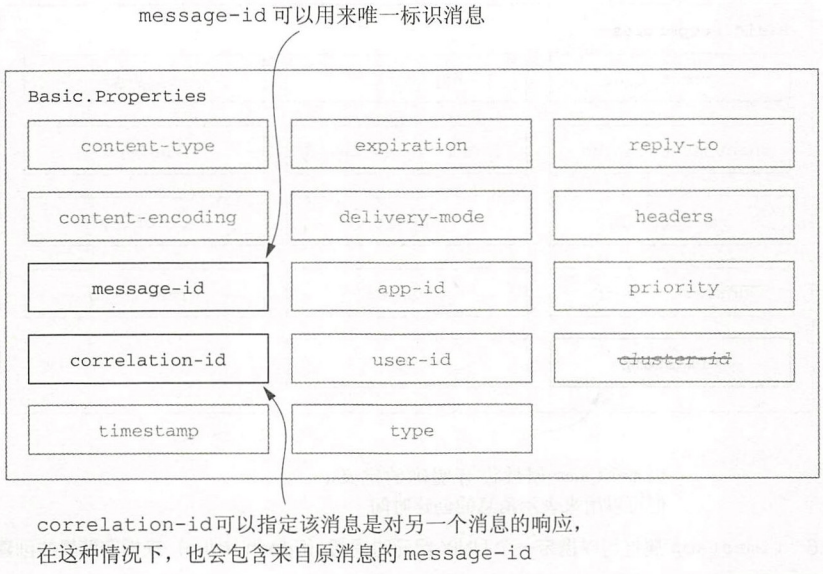


图 3.5　message-id 和 correlation-id 属性可用于跟踪单个消息及其响应消息在系统中的流动情况

3.4.1　Message-id

某些消息类型（如登录事件）并不需要与其关联的唯一标识，但我们很容易想象如销售订单或支持类请求等的消息需要具备这个唯一标识。当消息流经松耦合系统中的各个组件时，message-id 属性使得消息能够在消息头中携带数据，该数据可以唯一地识别该消息。

3.4.2　Correlation-id

虽然在 AMQP 规范中没有关于 correlation-id 的正式定义，但它的一个用途是指定该消息是另一个消息的响应，通过携带关联消息的 message-id 可以做到这一点。另一种选择是使用它来传送关联消息的事务 ID 或其他类似数据。

3.5 创建时间：timestamp 属性

Basic.Properties 中更有用的字段之一是 timestamp 属性（见图 3.6）。与 message-id 和 correlation-id 一样，timestamp 被指定为“应用级别使用”。即使你的消息没有使用它，timestamp 属性在你试图诊断经由 RabbitMQ 消息流中发生的任何意外行为时非常有用。通过使用 timestamp 属性来指示消息的创建时间，消费者可以评估消息投递过程的性能。

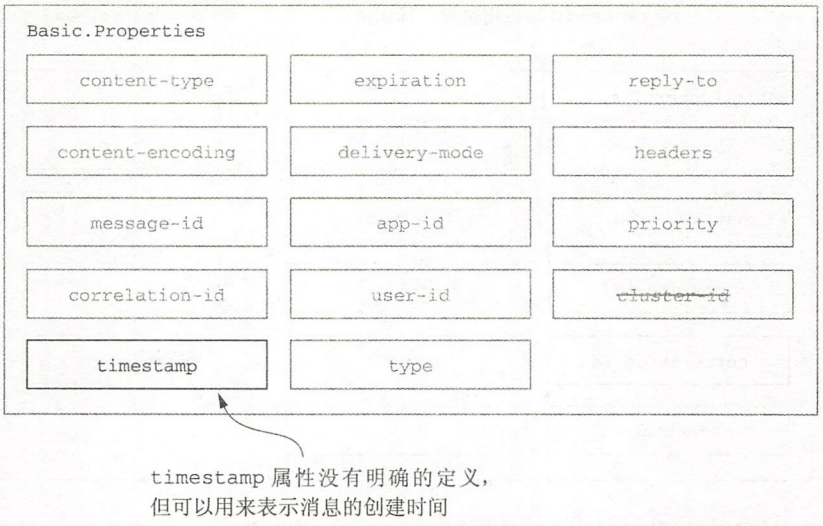


图 3.6 timestamp 属性可以携带一个 UNIX 纪元时间值（Epoch Value）来指定消息的创建时间

你的流程是否需要执行服务级别协议（Service Level Agreement，SLA）？通过判断消息属性中的 timestamp，消费者应用程序可以决定是否处理消息、丢弃消息，甚至向监控应用程序发布警报消息，以便让其他人知道消息的生存时间已经超过预期值。

所发送的时间戳可以是一个 UNIX 纪元时间，又或者是一个整数类的时间戳指示自 1970 年 1 月 1 日午夜以来的秒数。例如，2002 年 2 月 2 日的午夜将被表示为整数值 1329696000。作为一个编码之后的整数值，时间戳只占用 8 个字节的消息开销。不幸的是，时间戳没有时区上下文，因此建议在所有消息中使用 UTC 或其他统一的时区。当你的消息跨越多个时区到达地理位置分散的 RabbitMQ 代理服务器时，通过预先对时区进行标准化可以解决可能由此产生的后续问题。

3.6　消息自动过期

如果消息没有被消费，expiration 属性告诉 RabbitMQ 何时应该丢弃消息。尽管 AMQP 规范的 0-8 和 0-9-1 版本都存在 expiration 属性（见图 3.7），但在 3.0 版本发布之前，RabbitMQ 并不支持该属性。另外，expiration 属性的规范定义有点奇怪，它被指定为“用于实现，但没有正式的行为”，这意味着 RabbitMQ 可以提供它认为合理的任何实现方式。最后一个奇怪之处是它的格式是一个短字符串，最多允许 255 个字符，而代表时间单位的另一个属性 timestamp 则是一个整数值。

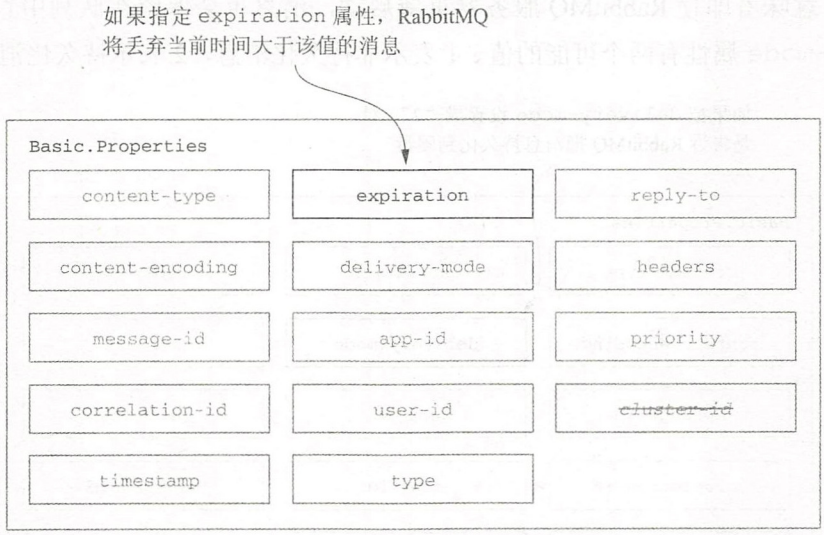


图 3.7　要在 RabbitMQ 中使用 expiration 属性，请将字符串值设置为一个 UNIX 纪元时间戳，用于指定消息的最大存活时间

由于规范中没有给出明确说明，当使用不同的消息代理服务器甚至同一消息代理服务器的不同版本时，expiration 值可能会有不同的含义。想要利用 expiration 属性来实现 RabbitMQ 消息的自动过期，它必须包含一个 UNIX 纪元时间或整数时间戳，然后把它存储为字符串。必须将字符串值设置为类似“1329696000”的格式，而不是存储像“2002-02-20T00：00：00-00”这样的 ISO-8601 格式的时间戳。

使用 expiration 属性时，如果把一个已经过期的消息发布到服务器，则该消息不会被路由到任何队列，而是直接被丢弃。

另外值得一提的是，RabbitMQ 还有可以让消息过期的其他功能，这些功能只在某些场景下才会生效。在声明一个队列时，你可以将一个 x-message-ttl 参数和队列定义一起

进行传递。这个值也应该是一个 UNIX 纪元时间戳，但是它使用毫秒精度（值 ×1000）来表示一个整数值。该值告诉队列在指定时间过后自动丢弃消息。x-message-ttl 队列参数及其优点将在第 5 章详细讨论。

3.7 使用 delivery-mode 平衡速度和安全性

delivery-mode 属性是一个字节字段，向消息代理服务器表明在将消息投递给任何正在等待的消费者之前，你希望先将它持久化到磁盘上（见图 3.8）。在 RabbitMQ 中，持久化消息意味着即使 RabbitMQ 服务器重新启动，消息也会保留在队列中直到被消费。delivery-mode 属性有两个可能的值：1 表示非持久化消息，2 表示持久化消息。

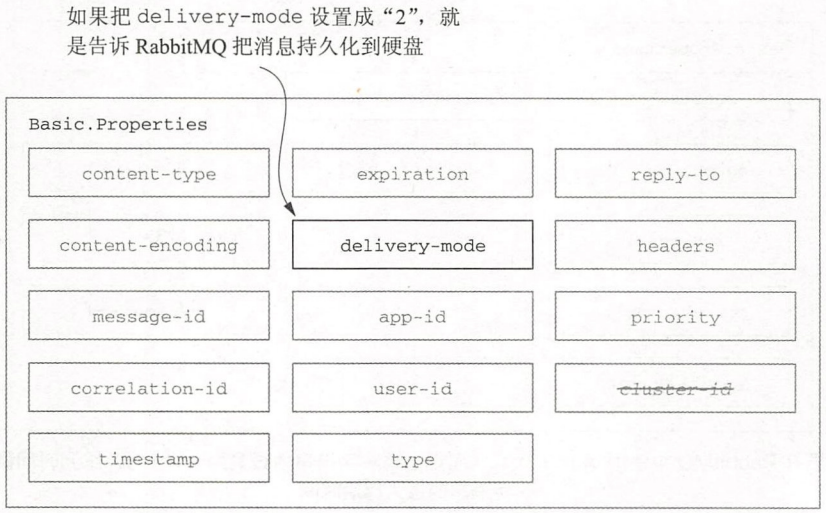


图 3.8 delivery-mode 属性指定 RabbitMQ 将消息放入队列时需要把消息保存在磁盘上，还是只保存在内存中

注意 当你第一次学习 RabbitMQ 中的各种术语和设置时，消息持久化通常可能会与队列中的持久性（durable）设置相混淆。队列的持久性属性告诉 RabbitMQ 队列的定义在重新启动 RabbitMQ 服务器或群集之后是否仍然有效。只有消息的 delivery-mode 才会向 RabbitMQ 指定消息是否应该被持久化。一个队列可能包含持久化和未持久化的消息。关于队列持久性将在第 4 章中讨论。

如图 3.9 所示，将消息设置为非持久化模式将允许 RabbitMQ 使用纯内存队列。

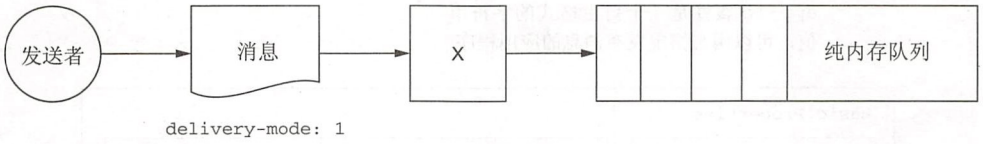


图 3.9　发布消息到纯内存队列中

由于内存 IO 本身比磁盘 IO 快，因此将 delivery-mode 指定为 1 将会尽可能降低消息投递的延迟性。在我的 Web 应用程序登录场景中，投递模式的选择可能比其他场景更容易。尽管我们希望 RabbitMQ 服务器发生故障时不会丢失任何登录事件，但这通常不是一个强需求。如果会员登录事件数据丢失了，业务也不太可能受到影响。在这种情况下，我们可以将 delivery-mode 设置为 1。但是，如果你使用 RabbitMQ 发布金融交易数据，并且你的应用程序架构专注于保证消息的可靠投递而不是消息吞吐量，那么你可以通过设置 delivery-mode 为 2 来启用持久化。如图 3.10 所示，当指定投递模式为 2 时，消息被保存到一个支持磁盘存储的队列中。

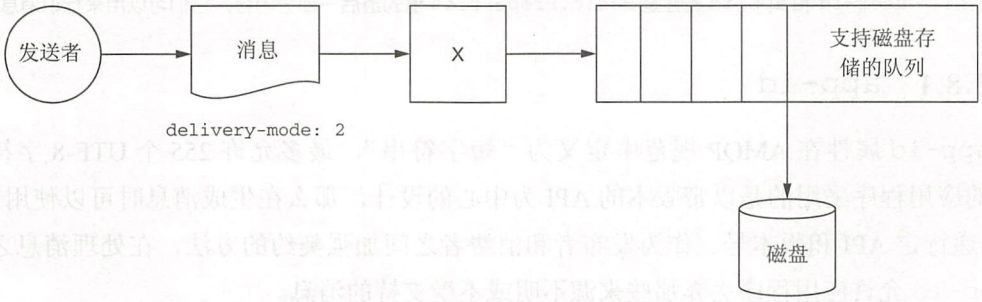


图 3.10　发布消息到支持磁盘存储的队列中

尽管这提供了一些保证，即当消息代理服务器崩溃时消息不会丢失，但是它存在潜在的性能和伸缩性问题。delivery-mode 属性对消息投递和性能影响巨大，具体会在第 4 章中详细展开。

3.8　使用 app-id 和 user-id 验证消息来源

app-id 和 user-id 属性提供了关于消息的另一层信息，并且有很多潜在的用途（见图 3.11）。如同其他可用于在消息中指定行为契约的属性一样，这两个属性也可以携带一些信息以便消费者应用程序在处理消息之前进行验证。

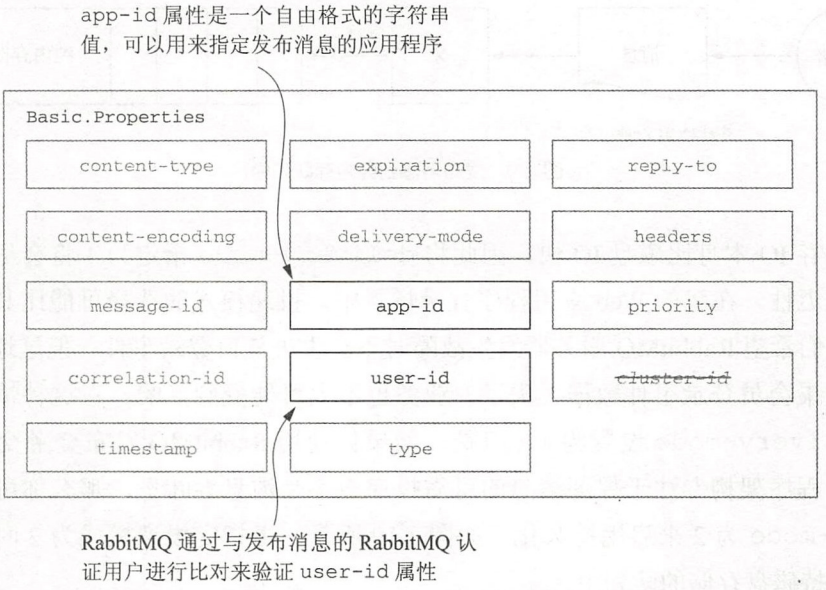


图 3.11　user-id 和 app-id 属性是 Basic.Properties 值的最后一部分内容，它们可以用来标识消息源

3.8.1　app-id

app-id 属性在 AMQP 规范中定义为“短字符串”，最多允许 255 个 UTF-8 字符。如果你的应用程序采用的是以带版本的 API 为中心的设计，那么在生成消息时可以使用 app-id 传递特定 API 和版本号。作为发布者和消费者之间加强契约的方法，在处理消息之前检查 app-id 允许应用程序丢弃那些来源不明或不受支持的消息。

app-id 的另一个用途是收集统计数据。例如，如果你使用消息来传递登录事件，则可以将 app-id 属性设置为触发登录事件的平台和应用程序版本。在一个需要同时支持 Web 端、桌面端和移动端应用的环境中，这将是透明地执行契约并提取数据以便跟踪平台登录的绝佳方式，在这种方式下我们甚至无须检查消息体。如果你希望专门用来收集统计信息的消费者能够和处理登录的消费者一样接收相同的消息，那么这将会非常方便。通过提供 app-id 属性，收集统计信息的消费者不必对消息体进行反序列化或解码。

提示　当试图追踪队列中的恶意消息时，加强使用 app-id 可以更容易地追踪恶意消息的来源。在许多应用程序共享 RabbitMQ 基础设施的大环境中，这点特别有用，

一个新的消息发布者可能会错误地使用与现有发布应用程序相同的交换器和路由键。

3.8.2　user-id

在用户身份验证的应用场景中，使用 user-id 属性来标识已登录的用户非常常见，但在大多数情况下这种做法并不推荐。RabbitMQ 会根据发布消息的 RabbitMQ 用户信息检查每条已发布消息的 user-id 属性值，如果这两个值不匹配，则该消息被拒绝。例如，如果你的应用程序在 RabbitMQ 中的认证用户名为“www”，而消息中的 user-id 属性值被设置为“linus”，则该消息将被拒绝。

当然，如果你的应用程序像聊天室或即时消息通信服务一样，你可能希望 RabbitMQ 中的用户为应用程序的每个用户所使用，并且确实希望使用 user-id 来识别实际登录到应用中的用户。

3.9　使用 type 属性获取明细

AMQP 规范的 0-9-1 版本将 Basic.Properties 中的 type 属性定义为“消息类型名称”，表示它用于应用程序并且没有规定正式的行为（见图 3.12）。尽管通过结合 routing-key 值与 exchange，通常可以传输足够多的信息用来确定消息内容，但 type 属性提供了另外一个工具，你的应用程序可以使用它来确定如何处理一个消息。

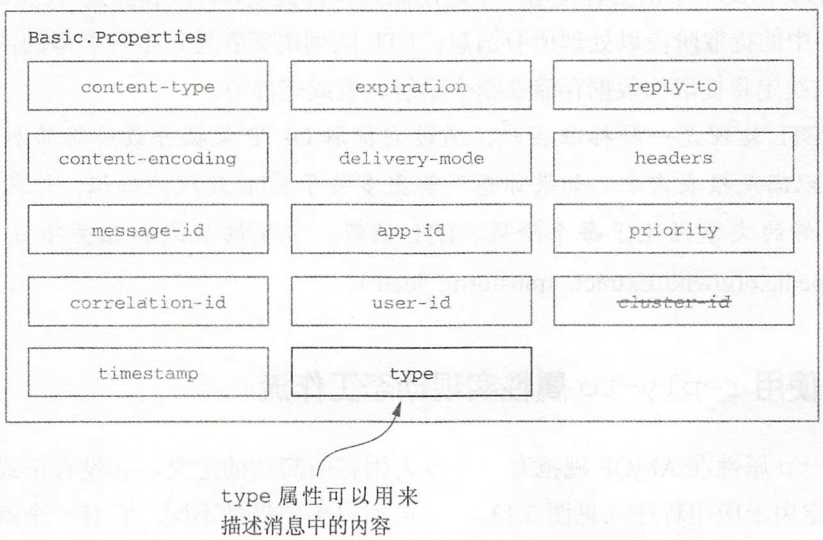


图 3.12　type 属性是一个可用于定义消息类型的自由格式字符串值

当自描述性的序列化格式不够快时

创建自描述消息时，type 属性非常有用，特别是当消息体没有以自描述数据格式进行序列化时。像 JSON 和 XML 这样的自描述格式被一些人认为太冗长了。它们也可能在网络传输或内存存储上带来不必要的开销，序列化和反序列化速度相较一些语言也比较慢。如果你正在考虑以上因素，你可以选择 Apache Thrift（http://thrift.apache.org/）或 Google 的 Protobuf（https://code.google.com/p/protobuf/）这样的序列化格式。与 MessagePack（http://msgpack.org/）不同，这些二进制编码的消息格式不是自描述的，需要依赖外部定义文件来进行序列化和反序列化。这种外部的依赖性和缺乏自描述性能够使网络传输上的有效载荷更小，但对其自身而言也是一种折衷方案。

当试图创建自描述 AMQP 消息用于加强发布者和消费者之间的契约时，在确定消息是否可以由消费者处理之前，非自描述的消息有效载荷需要被反序列化。在这种情况下，type 属性可用于指定记录类型或外部定义文件，如果无法正确访问处理消息所需的 .thrift 或 .proto 文件，消费者就能够拒绝这些消息。

在我提供的会员登录事件例子中，当将事件存储在数据仓库中时，我们发现将消息类型与消息一起使用非常有用。为了准备事件以便能够存储到数据仓库中，事件首先被存储在一个临时位置，然后用一个批处理程序读取它们并将它们存储到数据库中。由于这是一个非常通用的过程，因此单个消费者使用一个通用队列执行提取、转换和加载（extract-transform-load，ETL）中的提取阶段以处理所有消息。ETL 队列消费者处理多种类型的消息，并使用 type 属性来决定将提取的数据存储在哪个系统、表或集群中。

注意 ETL 处理是一种标准实践，通过它提取 OLTP 数据并最终将其加载到数据仓库以满足报表需求。如果你想了解更多关于 ETL 处理的知识，维基百科有一篇很好的文章描述了每个阶段、ETL 性能、常见的挑战和相关项目（http://en.wikipedia.org/wiki/Extract,_transform,_load）。

3.10 使用 reply-to 属性实现动态工作流

reply-to 属性在 AMQP 规范有一个令人困惑而简洁的定义，它没有正式规定行为，同时也被指定用于应用程序（见图 3.13）。与前面提到的属性不同，它有一个附加说明：使用 reply-to 可以构建一个用来回复消息的私有响应队列。尽管在 AMQP 规范中没有说明

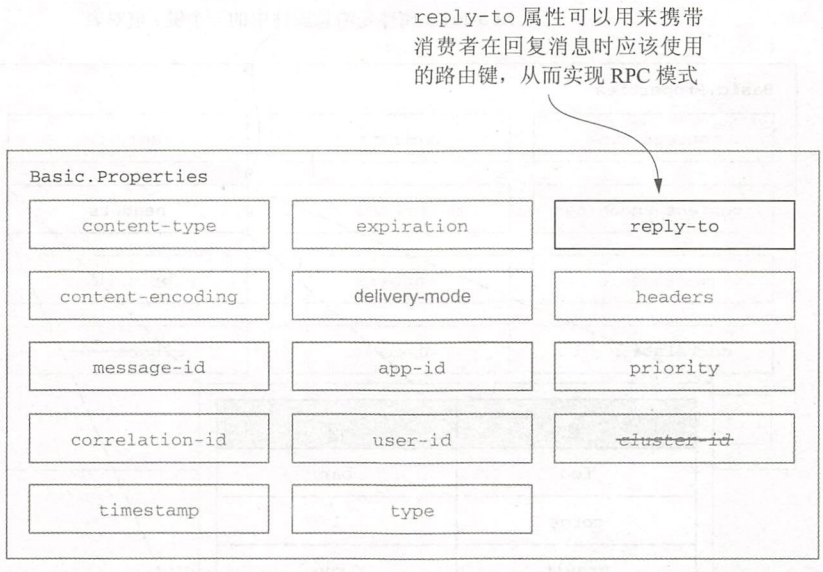


图 3.13　reply-to 属性没有正式的定义，但可以携带一个可用于回复消息的路由键或队列名

私有响应队列的确切定义，但是该属性可以很容易地在最初发布消息的相同交换器中携带特定的队列名称或路由键，这些队列名称或路由键可以用于回复消息。

警告　在 AMQP 规范的 0-9-1 版本中对 reply-to 属性有一个附加说明："在请求消息中使用私有响应队列时，它可以保存私有响应队列的名称"。这个定义中有太多的不明确性，所以应该谨慎使用这个属性。虽然未来版本的 RabbitMQ 不太可能在发布消息时强制执行响应消息的路由，但我们还是安全至上。考虑到 RabbitMQ 中 user-id 属性的行为规范以及这个属性的不明确性，如果响应消息由于 reply-to 属性中的信息而无法路由，那么 RabbitMQ 拒绝发布消息是合理的。

3.11　使用消息头自定义属性

headers 属性是一个键 / 值对表，允许用户自定义任意的键和值（见图 3.14）。键可以是 ASCII 或 Unicode 字符串，最大长度为 255 个字符。而值可以是任何有效的 AMQP 值类型。

与其他属性不同，headers 属性允许你添加任何你想要添加的数据到消息头表中。它还具有另一个独特的功能：RabbitMQ 可以根据 headers 表中填充的值路由消息，而不需要依赖于路由键。在第 6 章中会介绍通过 headers 属性进行消息路由。

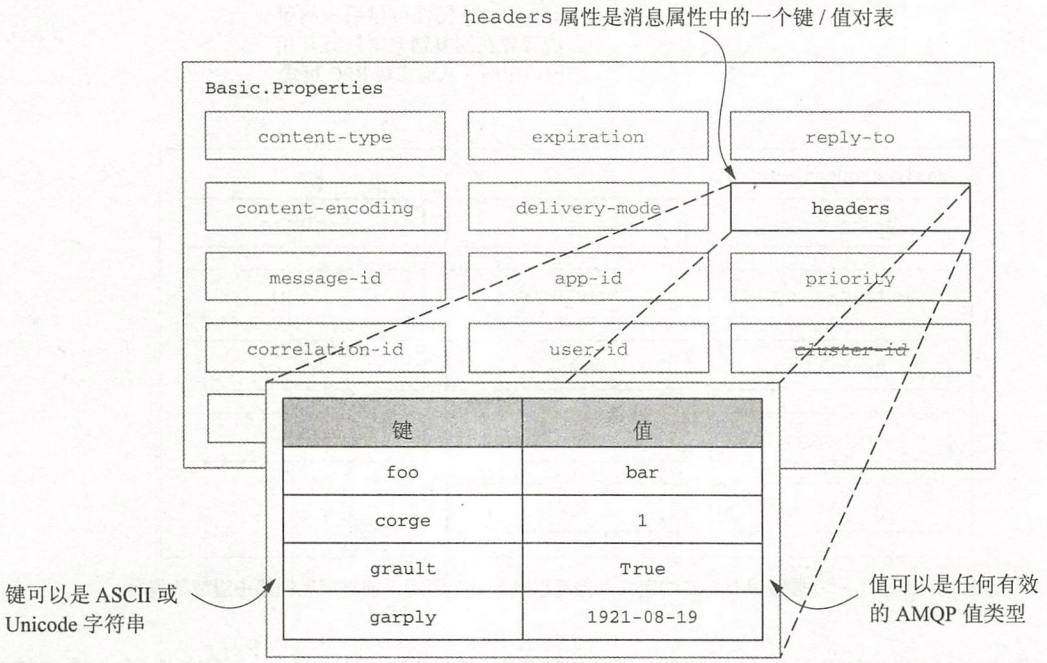


图 3.14 headers 属性允许消息属性中的任意键 / 值对

3.12 优先级属性

截至 3.5.0 版本，RabbitMQ 已按照 AMQP 规范实现了 priority 字段。它的可能值被定义为一个介于 0 到 9 之间的整数，用于指定队列中的消息优先级。按照规范，如果首先发布一条优先级为 9 的消息，随后再发布一条优先级为 0 的消息，则新连接的消费者将在优先级为 9 的消息之前接收到优先级为 0 的消息。有趣的是，RabbitMQ 将 priority 字段实现为无符号字节，所以优先级可以是 0 到 255 之间的任意值，但优先级应该被限制在 0 到 9 之间以保持与规范的互操作性，见图 3.15。

3.13 不能使用的属性：cluster-id/reserved

只有一个属性需要引起你的注意，只是为了让你知道不能使用它。你很可能注意到前面图（指图 3.2 ～图 3.15）中划掉的 cluster-id 属性（见图 3.16）。

cluster-id 属性是在 AMQP 0-8 中定义的，但随后被删除，RabbitMQ 从未实现过关于该属性的任何类型的行为。AMQP 0-9-1 将其重命名为 reserved，并声明它必须为空。虽然 RabbitMQ 目前没有根据规范要求它是空的，但你最好完全规避它。

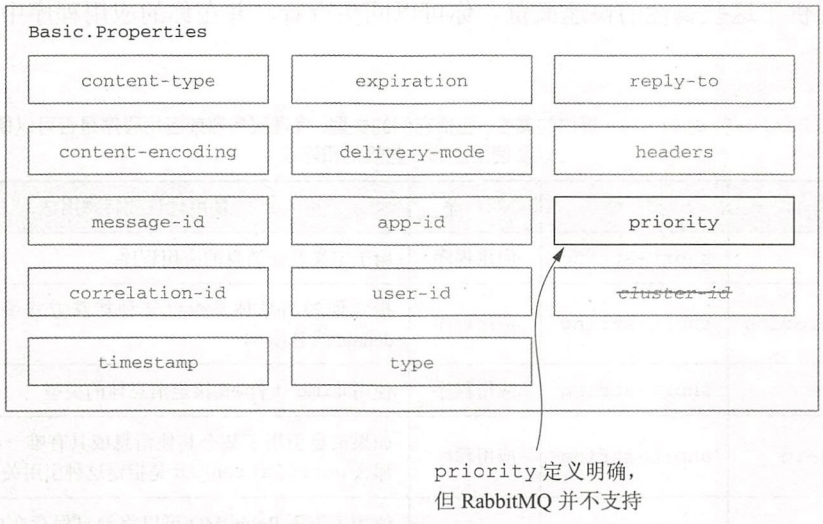


图 3.15　priority 属性可以用来指定队列中消息的优先级

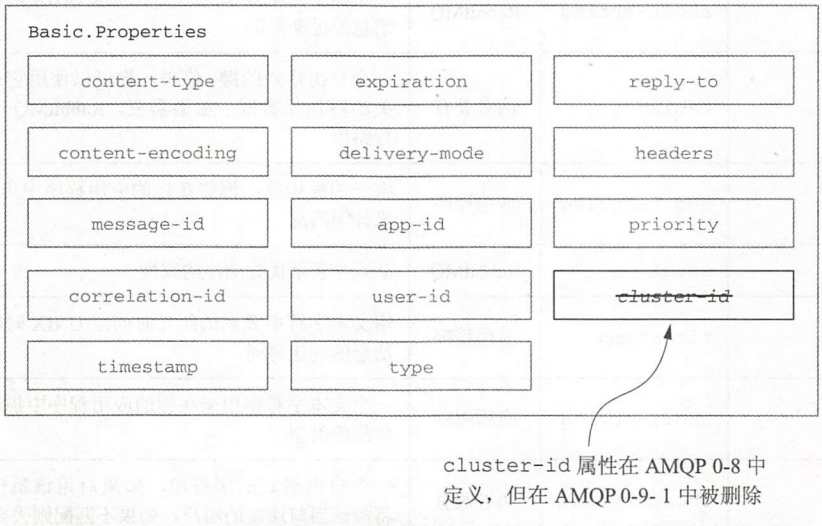


图 3.16　cluster-id 属性被重命名并保留在 AMQP 0–9– 1，我们不应该使用它

3.14　小结

通过正确使用 Basic.Properties，你的消息通信架构可以在发布者和消费者之间创建严格的行为契约。此外，它们能够使你的消息适应未来的项目集成需求，这些需求可能在初始应用程序和消息规范中并没有考虑到。

表 3.1 提供了这些属性的快速概览。你可以回头查看，并在你的应用程序中合理应用这些属性。

表 3.1 由 Basic.Properties 提供的属性，包括它们的类型、代理服务器或应用程序是否可以使用它们，以及使用上的一些规范和建议

属 性	类 型	用 途	使用建议或特殊用法
app-id	short-string	应用程序	用于定义发布消息的应用程序
content- encoding	short-string	应用程序	指定你的消息体是否以某种特殊方式编码，如 zlib、deflate 或 Base64
content-type	short-string	应用程序	使用 mime-types 指定消息体的类型
correlation-id	short-string	应用程序	如果消息引用了某个其他消息或具有唯一标识的项目，那么 correlation-id 是指定这种引用关系的有效途径
delivery-mode	octet	RabbitMQ	值为 1 告诉 RabbitMQ 可以将消息保存在内存中；值为 2 表示它也应该被写入磁盘
expiration	short-string	RabbitMQ	用文本字符串表示的纪元时间或 UNIX 时间戳值，表示消息的过期时间
headers	table	两者兼有	一个自由格式的键 / 值表，你可以使用它来添加消息相关的附加元数据；如果需要，RabbitMQ 可以基于它进行路由
message-id	short-string	应用程序	唯一的标识符，例如在你的应用程序中可以使用 UUID 来标识消息
priority	octet	RabbitMQ	队列中表示优先顺序的属性
timestamp	timestamp	应用程序	用文本字符串表示的纪元时间或 UNIX 时间戳值，表示消息的创建时间
type	short-string	应用程序	一个文本字符串用来在你的应用程序中描述消息或有效负载的类型
user-id	short-string	两者兼有	一个自由格式的字符串，如果启用该属性，RabbitMQ 将验证当前连接的用户，如果不匹配则丢弃消息

除了使用自描述消息的属性之外，这些属性还可以携带对消息有价值的元数据，这些元数据将允许你创建复杂的路由和事务机制，而不必使用与消息相关的上下文信息来污染

消息体。在评估进行投递的消息时，RabbitMQ 将利用特定的属性（例如 delivery-mode 和 headers 表）来确保你的消息按照你指定的方式和目的地进行投递。但是，为了确保消息投递万无一失，这些属性值只是冰山一角。

第 4 章　消息发布的性能权衡

本章概要：

- RabbitMQ 中的消息可靠投递
- 发布者 vs 性能权衡

消息发布是消息通信架构的核心活动之一，在 RabbitMQ 中，消息发布涉及很多方面。应用程序可以使用很多消息发布选项，这些选项可能会对应用程序的性能和可靠性产生巨大影响。虽然性能和吞吐量可以用来衡量任何消息代理服务器，但是我们最关注的是消息的可靠投递。试想一下，在你使用自动取款机将资金存入你的银行账户时，如果这个过程没有保障的话会发生什么情况。你将无法确定你的账户余额会增加。这将不可避免地成为你和银行之间的一个问题。即使在非核心应用程序中，消息也是为了某种目的而发布的，悄无声息地将它们丢弃很容易产生问题。

虽然并不是每个系统都有像银行应用程序一样对消息可靠投递有如此严格的要求，但确保消息被接收和投递对像 RabbitMQ 这样的软件而言是重要的。AMQP 规范提供消息发布中的事务以及消息持久化选项，以提供比自身普通消息发布更高级别的可靠消息通信机制。

RabbitMQ 还具备其他功能，例如投递确认功能，该功能提供了不同级别的消息可靠投递机制供你选择，包括跨越多个服务器的高可用性（Highly Available，HA）队列。在本章中，你将了解使用这些功能所涉及的性能和可靠发布之间的权衡，以及如何查明 RabbitMQ 是否正在悄悄地对你的消息发布者进行限流。

4.1　平衡投递速度与可靠投递

就 RabbitMQ 而言，金发姑娘原则（Goldilocks Principle）适用于消息投递在不同级别上的保障机制。金发姑娘原则摘自《三只熊的故事》一书，描述了什么是刚刚好（just right）。在 RabbitMQ 中实现可靠消息投递的情况下，你应该将此原则应用在使用可靠投递机制时所遇到的平衡性问题。有些功能可能对你的应用程序来说太慢了，比如确保消息在 RabbitMQ 服务器重新启动后仍然存在。另一方面，发布消息而不要求额外的保障机制会快得多，尽管它可能无法为重要的应用程序提供足够安全的环境（见图 4.1）。

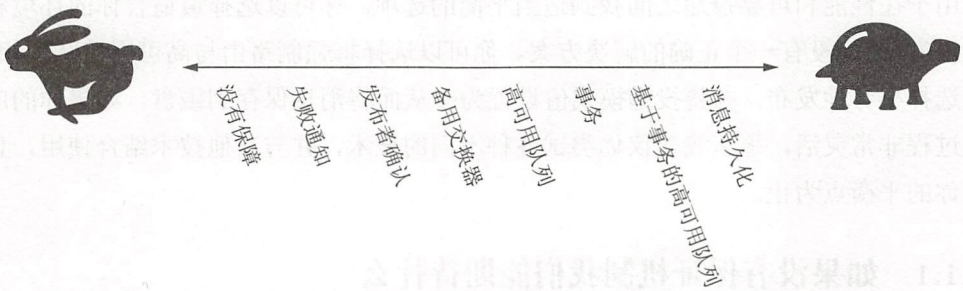


图 4.1　使用每个可靠投递机制时性能将会受到影响，当组合使用时更是如此

在 RabbitMQ 中，旨在创建可靠投递的每个机制都会对性能产生一定的影响。单独使用时，你可能不会注意到吞吐量的显著差异，但是当它们组合使用时，可能会对消息吞吐量产生重大影响。只有通过执行自己的性能基准测试，才能确定性能与可靠投递之间可以接受的平衡。

在使用 RabbitMQ 创建应用程序体系结构时，应该牢记金发姑娘原则。下面的问题可以帮助找到刚刚好的解决方案，用于平衡高性能和消息传递的安全性。

- 消费发布时保证消息进入队列的重要性有多高？
- 如果消息无法路由，是否应将消息返回给发布者？

- 如果消息无法路由，是否应该将其发送到其他地方稍后进行重新路由？
- 如果 RabbitMQ 服务器崩溃，可以接受信息丢失吗？
- RabbitMQ 在处理新消息时是否应该确认它已经为发布者执行了所有请求的路由和持久化任务？
- 消息发布者是否可以批量投递消息，然后从 RabbitMQ 收到一个确认用于表明所有请求的路由和持久化任务已经批量应用到所有的消息中？
- 如果你要批量发布消息，而这些消息需要确认路由和持久化，那么对每一条消息是否需要对目标队列实现真正意义上的原子提交？
- 在可靠投递方面是否有可接受的平衡性，你的发布者可以使用它来实现更高的性能和消息吞吐量吗？
- 消息发布还有哪些方面会影响消息吞吐量和性能？

在本节中，我们将讨论这些问题与 RabbitMQ 的关系，以及你的应用程序可以采用哪些技术和功能来实现恰到好处的可靠投递和性能。在本章的行文思路中，你将看到 RabbitMQ 提供的用于在性能和可靠投递之间找到适当平衡的选项。你可以选择最适合你的环境和应用程序的内容，因为没有一个正确的解决方案。你可以选择将强制路由与高可用性队列相结合，也可以选择事务性发布，并将投递模式值设置为 2 从而将消息保存到磁盘。如果你的应用程序开发过程非常灵活，那么我建议你尝试各种不同的技术，并与其他技术结合使用，直到找到适合你的平衡点为止。

4.1.1 如果没有保证机制我们能期待什么

在完美的世界中，无须任何额外的配置或步骤，RabbitMQ 就能可靠地投递消息。只需通过 `Basic.Publish` 发布你的消息并使用正确的交换器和路由信息，你的消息会被接收并发送到适当的队列。没有网络问题，服务器硬件可靠而不会崩溃，操作系统永远不会出现影响 RabbitMQ 代理服务器运行时状态的问题。运行在乌托邦式的应用环境中，通过与可能会减慢它们处理速度的服务进行交互，你的消费者应用程序将永远不会面临性能上的限制。队列永远不会阻塞，消息的处理速度与发布一样快。消息发布不会面临任何形式的限流。

不幸的是，在把墨菲法则（Murphy's Law）当作一个经验法则的世界里，那些永远不会发生在完美世界中的事情则会经常发生。

在非核心应用程序中，正常的消息发布不必处理每个可能的故障点；找到合适的平衡将

使你获得可靠和可预测的运行时间。在一个形成闭环的环境中，你不必担心网络或硬件故障，也不必担心消费者的消费速度不够快，RabbitMQ 的架构和功能集展示的可靠消息通信级别足以满足大部分非核心类应用程序。例如，最初由 Orbitz 开发的具备高度可扩展的图形系统 Graphite 有一个 AMQP 接口用于将统计数据提交到 Graphite。运行测量数据收集服务（如 collectd）的独立服务器收集有关其运行时状态的信息并按每分钟一次的频率发布消息（见图 4.2）。

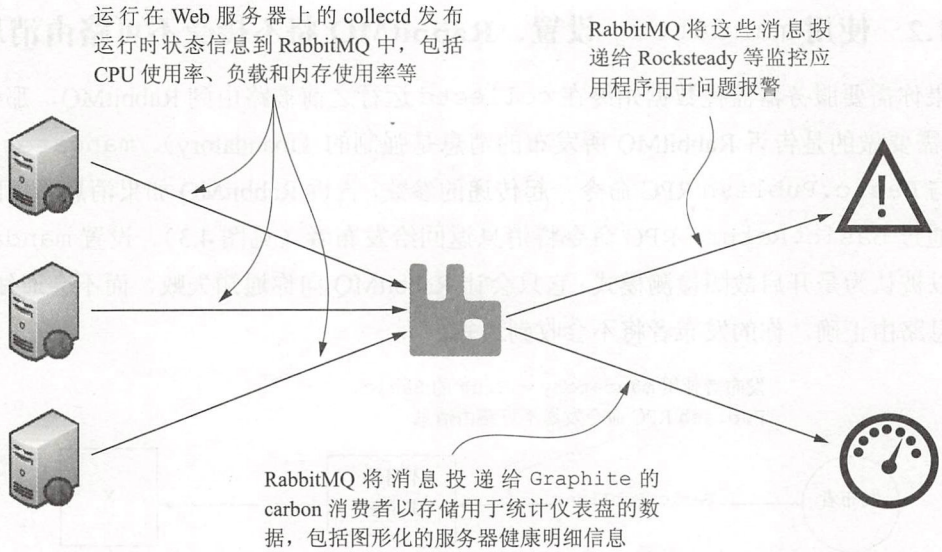


图 4.2　Web 服务器 collectd 的统计信息收集守护进程将监控数据发布到 RabbitMQ，以便投递给 Graphite 和 Rocksteady 消费者

这些消息携带诸如 CPU 负载、内存和服务器网络利用率等信息。Graphite 有一个名为 carbon 的收集器服务，它会消费这些消息并将数据存储在其内部数据存储媒介中。在大多数环境中，即使这些数据在网络的整体运营管理中可能非常重要，它们也会被认为是非核心数据。就算一分钟的数据没有被 carbon 收集并存储在 Graphite 中，也不会被认为是失败，这点与金融事务的处理级别不同。缺少样本数据实际上可能表明服务器本身或者将数据发布到 Graphite 的过程出现了问题，而 Rocksteady 等系统可以使用此类数据在 Nagios 或其他类似的应用程序中触发事件，从而对问题进行报警。

在发布这样的数据时，你需要了解其中的平衡性。在没有额外可靠发布机制的情况下传递监控数据，配置项可以更少，处理开销也更低，并且比可靠消息投递更简单。在这种情况下，刚刚好是一个简单的设置，不使用额外的消息可靠投递机制。collectd 进程可以触发

并忘记它所发送的消息。如果它与 RabbitMQ 断开连接，它将在下次需要发送统计数据时尝试重新连接。同样，消费者应用程序在断开连接时将重新连接，并回到之前消费的同一个队列继续消费。

这在大多数情况下都能很好地工作，直到墨菲定律发挥作用，系统出现了问题。如果你希望确保你的消息始终都能投递成功，那么 RabbitMQ 可以更换装备，并且足够应对核心业务场景。

4.1.2 使用 mandatory 设置，RabbitMQ 将不接受不可路由消息

如果你需要服务器监控数据始终在 collectd 运行之前被路由到 RabbitMQ，那么所有 collectd 需要做的是告诉 RabbitMQ 所发布的消息是强制的（mandatory）。mandatory 标志是一个与 Basic.Publish RPC 命令一起传递的参数，告诉 RabbitMQ 如果消息不可路由，它应该通过 Basic.Return RPC 命令将消息返回给发布者（见图 4.3）。设置 mandatory 标志可以被认为是开启故障检测模式；它只会让 RabbitMQ 向你通知失败，而不会通知成功。如果消息路由正确，你的发布者将不会收到通知。

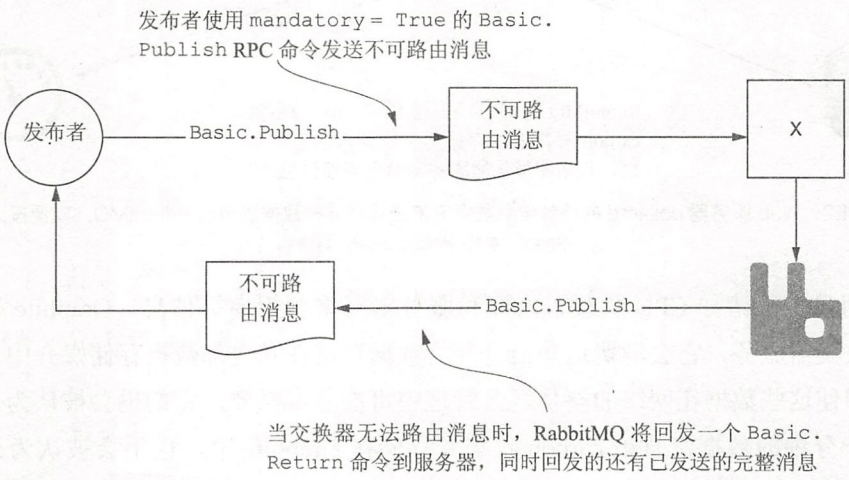
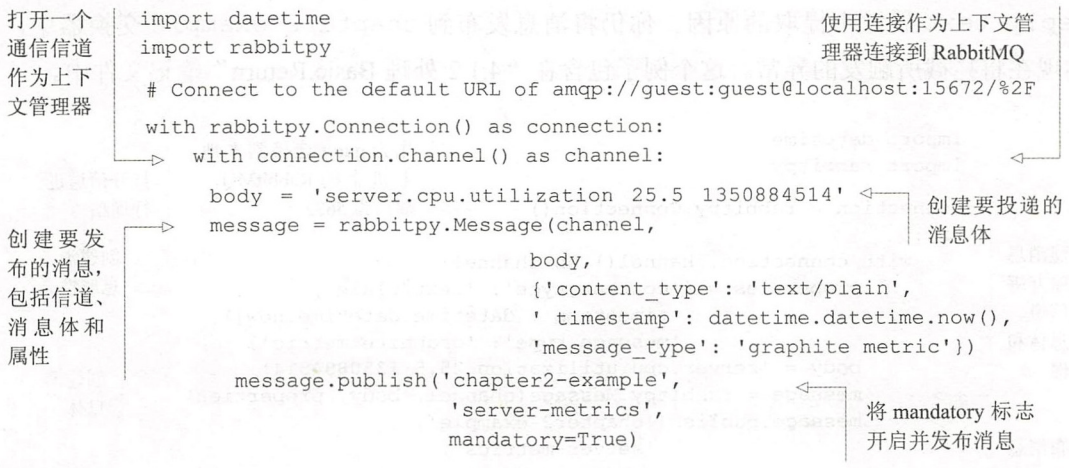


图 4.3 当以 mandatory = True 发布一个不可路由的消息时，RabbitMQ 通过 Basic.Return RPC 调用将消息返回给客户端

要发布带有 mandatory 标志的消息，只需在交换器、路由键、消息以及消息属性参数之后传入该标志，如以下示例所示。要触发不可路由消息的预期异常，可以使用与第 2 章中相同的交换器。当发布消息时没有绑定目标地址时，执行时应该就会引发异常。代码位于“4.1.2 发布失败”笔记文件中。




```
import datetime
import rabbitpy

# Connect to the default URL of amqp://guest:guest@localhost:15672/%2F
with rabbitpy.Connection() as connection:
    with connection.channel() as channel:

        body = 'server.cpu.utilization 25.5 1350884514'
        message = rabbitpy.Message(channel,
                                   body,
                                   {'content_type': 'text/plain',
                                    ' timestamp': datetime.datetime.now(),
                                    'message_type': 'graphite metric'})
        message.publish('chapter2-example',
                        'server-metrics',
                        mandatory=True)
```

当你执行这个例子时，你会收到一个类似如下的异常。RabbitMQ 不能路由消息，因为没有队列绑定到交换器和路由键。

注意　在前面的例子中，使用了一个调用 Connection 和 Channel 对象的新方法：两个对象都被创建为上下文管理器（context manager）。在 Python 中，如果一个对象是一个上下文管理器，当你退出使用该对象的作用域或缩进级别时，它将自动关闭对象实例。在使用 rabbitpy 的情况下，当你退出作用域时，它将正确关闭信道和连接，而不必分别显式地调用 Channel.close 或 Connection.close。

Basic.Return 调用是一个 RabbitMQ 的异步调用，并且在消息发布后的任何时候都可能发生。例如，当 collectd 将统计数据发布到 RabbitMQ 时，如果发布失败，它可能会在收到 Basic.Return 调用之前发布多个数据点。如果代码没有为这个调用设置监听器，该调用就会被忽略，collectd 也就永远不会知道消息没有正确发布。如果你想确保消息被投递到正确的队列，那这就是一个问题。

在 rabbitpy 库中，Basic.Return 调用被客户端库自动接收，并在信道范围内收到该调用时就会触发 MessageReturnedException 异常。以下示例将使用相同的路由键将相同的消息发送到同一个交换器。用于发布消息的代码已做了轻微重构以将信道范围封装在 try/except 块中。当引发异常时，代码将打印消息 ID 并返回从 Basic.Return 帧的

reply-text 属性中提取的原因。你仍将消息发布到 chapter2-example 交换器中，那你现在将拦截所触发的异常。这个例子包含在“4.1.2 处理 Basic.Return”笔记文件中。

当你执行这个例子时，你应该看到一个更友好的消息，像这样：

```
Message was returned by RabbitMQ: (312) NO_ROUTE for exchange chapter2-example
```

与其他库一样，如果在发布消息时收到来自 RabbitMQ 的 Basic.Return RPC 调用，则可能需要注册一个回调方法以响应这个 RPC 调用。在实际处理 Basic.Return 消息的异步编程模型中，你将收到一个 Basic.Return 方法帧、内容标题帧以及消息体帧，就像你在消费消息一样。如果这看起来太复杂，不要担心，还有其他方法可以简化流程并处理消息路由失败的场景。其中一个是使用 RabbitMQ 的发布者确认（Publisher Confirms）。

注意 在发送 Basic.Publish 命令时，rabbitpy 库和本节中的示例最多只使用三个参数。这与 AMQP 规范不同，AMQP 规范包括一个额外的 immediate 标志符。如果消息不能立即路由到目的地，immediate 标志告诉代理服务器发出一个 Basic.Return。这个标志从 RabbitMQ 2.9 版开始不推荐使用，如果使用的话会引发一个异常并且关闭信道。

4.1.3 发布者确认作为事务的轻量级替代方法

RabbitMQ 中的发布者确认功能是 AMQP 规范的增强功能，只能用在支持 RabbitMQ 特定扩展的客户端库中。尽管在磁盘上存储消息是防止消息丢失的重要一步，但这样做并

不会在发布者和 RabbitMQ 服务器之间创建一个契约以告诉发布者消息已经投递成功。在发布任何消息之前，消息发布者必须向 RabbitMQ 发出 Confirm.Select RPC 请求，并等待 Confirm.SelectOk 响应以获知投递确认已经被启动。在这一点上，对于发布者发送给 RabbitMQ 的每条消息，服务器会发送一个确认响应（Basic.Ack）或否定确认响应（Basic.Nack），这两个响应都包括一个整数值用于指定确认消息的偏移值（见图 4.4）。确认编号通过 Confirm.Select RPC 请求之后的消息接收顺序来引用消息。

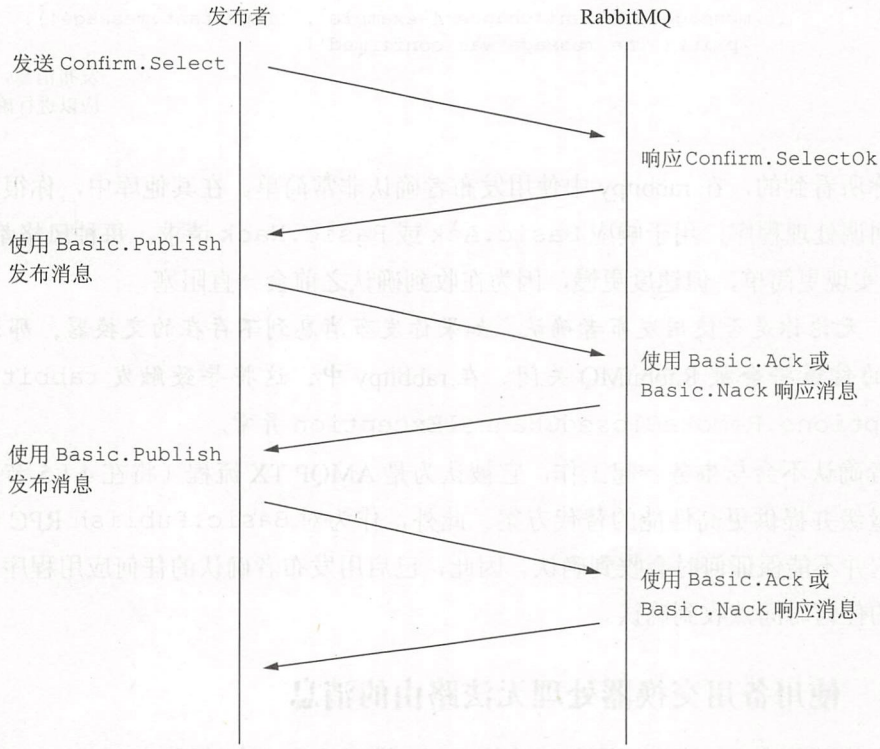
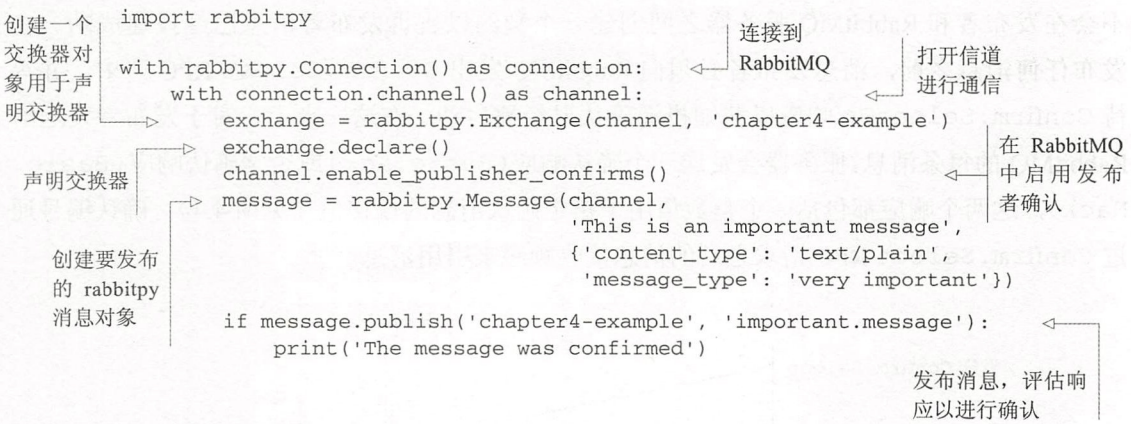


图 4.4　投递确定中，发送消息到 RabbitMQ 或从 RabbitMQ 接收消息的时序图

当发布者发布给所有队列的已路由消息被消费者应用程序直接消费时，或者消息被放入队列并根据需要进行持久化时，一个 Basic.Ack 请求会被发送到发布者。如果消息无法路由，代理服务器将发送一个 Basic.Nack RPC 请求用于表示失败。然后由发布者决定该如何处理该信息。

在“4.1.3 发布者确认”笔记文件中包含以下示例，发布者启用发布者确认，然后评估 Message.publish 调用的响应。




```
import rabbitpy

with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        exchange = rabbitpy.Exchange(channel, 'chapter4-example')
        exchange.declare()
        channel.enable_publisher_confirms()
        message = rabbitpy.Message(channel,
                                   'This is an important message',
                                   {'content_type': 'text/plain',
                                    'message_type': 'very important'})

        if message.publish('chapter4-example', 'important.message'):
            print('The message was confirmed')
```

正如你所看到的，在 rabbitpy 中使用发布者确认非常简单。在其他库中，你很可能需要创建一个回调处理程序，用于响应 `Basic.Ack` 或 `Basic.Nack` 请求。每种风格都有好处：rabbitpy 的实现更简单，但速度更慢，因为在收到确认之前会一直阻塞。

注意 无论你是否使用发布者确认，如果你发布消息到不存在的交换器，那么发布用的信道将会被 RabbitMQ 关闭。在 rabbitpy 中，这将导致触发 `rabbitpy.exceptions.RemoteClosedChannelException` 异常。

发布者确认不会与事务一起工作，它被认为是 AMQP TX 流程（将在 4.1.5 节中讨论）的一种轻量级并提供更高性能的替代方案。此外，作为对 `Basic.Publish` RPC 请求的异步响应，它并不能保证何时会收到确认。因此，已启用发布者确认的任何应用程序可以在发送消息后的任何时间点收到确认。

4.1.4 使用备用交换器处理无法路由的消息

备用交换器是由 RabbitMQ 团队创建的 AMQ 模型的另一个扩展，用于处理无法路由的消息。备用交换器在第一次声明交换器时被指定，用来提供一种预先存在的交换器，即如果交换器无法路由消息，那么消息就会被路由到这个新的备用交换器（见图 4.5）。

注意 如果在将消息发送到具有备用交换器的交换器上时设置了 `mandatory` 标志，那么一旦预期的交换器无法正常路由消息，`Basic.Return` 就不会发给发布者。当 `mandatory` 标志为 `true` 时，向备用交换器发送不可路由消息的行为满足消息已经被发布这一条件。RabbitMQ 的消息路由模式与其他交换器一样适用于

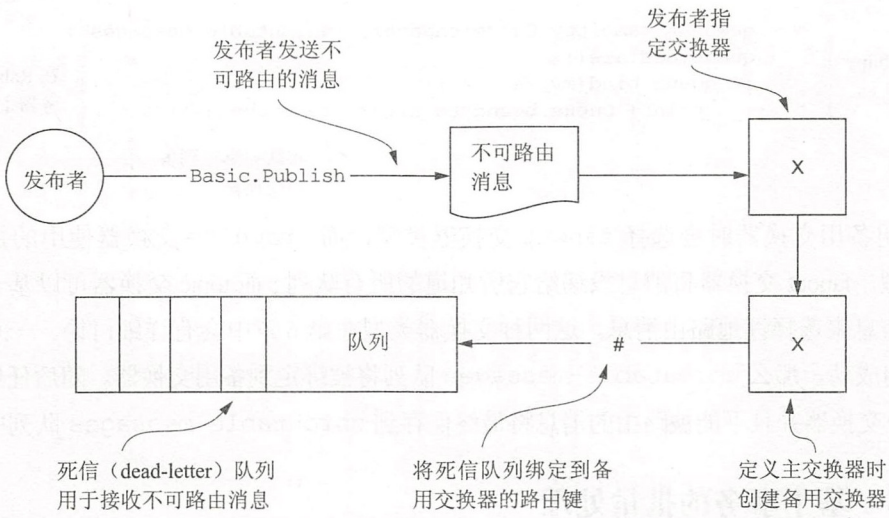


图 4.5　当一个不可路由的消息发布到一个已经定义了备用交换器的交换器中时，它将被路由到备用交换器

备用交换器，认识到这一点也很重要。如果一个队列没有使用它的初始路由键进行绑定以接收消息，那么该消息将不会被放入队列，并且将会丢失。

要使用备用交换器，你必须首先创建用来接收不可路由消息的交换器。然后，在设置接收消息的主交换器时，将 alternate-exchange 参数添加到 Exchange.Declare 命令中。下面的例子演示了这个过程，它进一步创建了一个消息队列用来存储所有不可路由的消息。这个例子在“4.1.4 备用交换器例子”笔记文件中。

```
import rabbitpy

with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        my_ae = rabbitpy.Exchange(channel,
                                  'my-ae',
                                  exchange_type='fanout')
        my_ae.declare()

        args = {'alternate-exchange': my_ae.name}

        exchange = rabbitpy.Exchange(channel,
                                     'graphite',
                                     exchange_type='topic',
                                     arguments=args)
        exchange.declare()
```

连接到 RabbitMQ

打开一个信道进行通信

为备用交换器创建一个 rabbitpy 交换器对象

在 RabbitMQ 服务器上声明交换器

为 graphite 交换器定义用于创建备用交换器的字典

使用 rabbitpy 交换器对象创建名为 graphite 的交换器，并传入 args 字典

声明 graphite 交换器

```
queue = rabbitpy.Queue(channel, 'unroutable-messages')
queue.declare()
if queue.bind(my_ae, '#'):
    print('Queue bound to alternate-exchange')
```

创建一个 rabbitpy 队列对象

在 RabbitMQ 服务器上声明队列

将队列绑定到备用交换器

在声明备用交换器时会选择 fanout 交换器类型，而 graphite 交换器使用的是 topic 交换器类型。fanout 交换器将消息投递给它所知道的所有队列；而 topic 交换器可以基于路由键中的部分信息来选择性地路由消息。这两种交换器类型在第 6 章中会有详细讨论。一旦这两个交换器声明成功，那么 unroutable-messages 队列将被绑定到备用交换器。随后任何发布到 graphite 交换器并且不能被路由的消息将最终保存到 unroutable-messages 队列中。

4.1.5 基于事务的批量处理

在投递确认出现之前，确保消息被成功投递的唯一方法是事务。AMQP 事务（也就是 TX）类提供了一种机制，通过这种机制，消息可以批量发布到 RabbitMQ，然后提交到队列或回滚。以下示例展示了编写事务代码以利用它的优势是非常简单的，该示例代码包含在“4.1.5 事务性发布”笔记文件中。

```
import rabbitpy

with rabbitpy.Connection() as connection:
    with connection.channel() as channel:

    tx = rabbitpy.Tx(channel)
    tx.select()

    message = rabbitpy.Message(channel,
                               'This is an important message',
                               {'content_type': 'text/plain',
                                'delivery_mode': 2,
                                'message_type': 'important'})
    message.publish('chapter4-example', 'important.message')
    try:
        if tx.commit():
            print('Transaction committed')
    except rabbitpy.exceptions.NoActiveTransactionError:
        print('Tried to commit without active transaction')
```

连接到 RabbitMQ

打开一个信道进行通信

启动事务

创建一个 rabbitpy.Tx 对象实例

创建用于发布的消息

发布消息

提交事务

捕获可能触发的 TX 异常

事务机制提供了一种方法，通过这种方法可以通知发布者消息被成功投递到 RabbitMQ 代理服务器上的队列。要启动一个事务，发布者发送一个 TX.Select RPC 请求给 RabbitMQ，RabbitMQ 将回复一个 TX.SelectOk 响应。一旦事务被打开，发布者可以向 RabbitMQ 发送一个或多个消息（见图 4.6）。

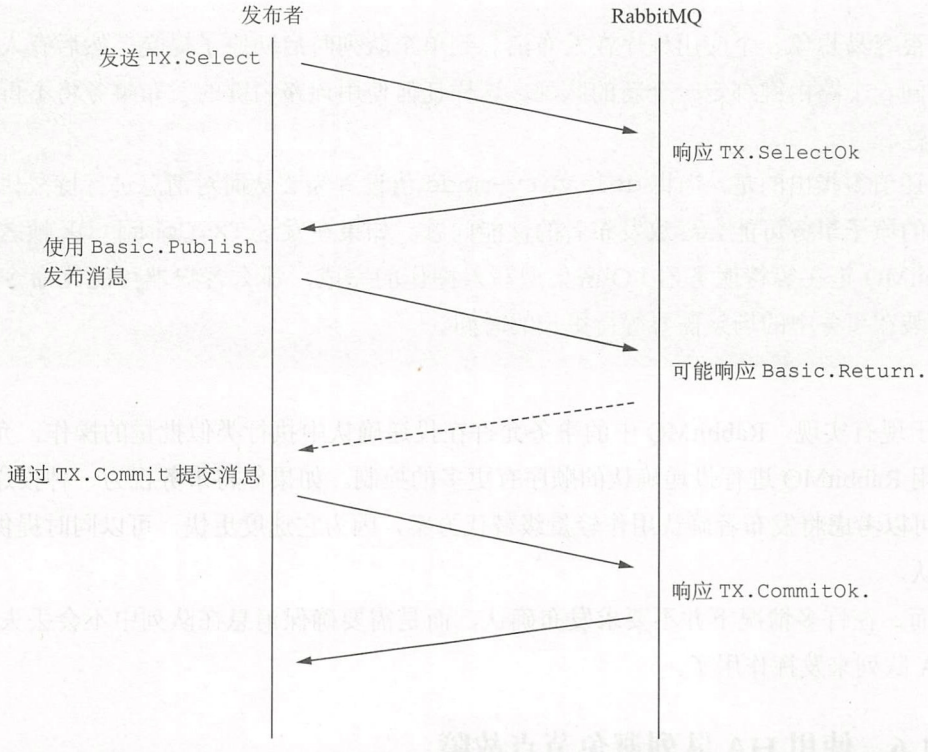


图 4.6　发布者通过发送 TX.Select 命令来启动事务和发布消息，并使用 TX.Commit 命令来提交消息

当 RabbitMQ 由于错误（例如不存在的交换器）而无法路由消息时，它将在发送 TX.CommitOk 响应之前返回一个带有 Basic.Return 响应的消息。希望中止事务的发布者应该发送 TX.Rollback RPC 请求并等待来自代理服务器的 TX.RollbackOk 响应，然后再继续后续的工作。

RabbitMQ 与原子事务

原子性确保事务中所有操作的完成都将作为事务提交的一部分。在 AMQP 中，这意味着直到事务中的所有操作都完成为止，你的客户端将不会收到 TX.CommitOk 响应帧。不幸的是，对于那些寻求真正原子性的人来说，RabbitMQ 只在每个发出的命令作用于单个队列时才执行原子事务。如果不止一个队列受到事务中任何命令的影响，则提交就不具备原子性。

尽管当事务中的所有命令仅影响同一个队列时 RabbitMQ 会执行原子事务，但发布者通常不能很好地控制消息是否被投递到多个队列。使用 RabbitMQ 的高级路由方

法，很容易想象一个应用程序在发布消息到单个队列时启动原子提交，然后有人可能使用同一个路由键绑定一个新的队列。这样任何使用该路由键的发布事务将不再具备原子性。

还值得指出的是，当将 delivery-mode 值设置为 2 从而对消息进行持久化时，真正的原子事务可能会导致发布者的性能问题。如果在发送 TX.CommitOk 帧之前，RabbitMQ 正在等待服务器 I/O 密集型写入操作的完成，那么客户端可能比命令没有被包装在事务中的场景需要等待更长的时间。

基于现有实现，RabbitMQ 中的事务允许在投递确认中执行类似批量的操作，允许发布者对使用 RabbitMQ 进行投递确认的顺序有更多的控制。如果你将事务视为一种投递确认的方式，可以考虑将发布者确认用作轻量级替代方案，因为它速度更快，可以同时提供肯定和否定确认。

然而，在许多情况下并不要求发布确认，而是需要确保消息在队列中不会丢失。这就轮到 HA 队列来发挥作用了。

4.1.6 使用 HA 队列避免节点故障

当你希望加强发布者与 RabbitMQ 之间的契约以保证消息投递时，请不要忽视高可用队列（HA 队列）在核心消息通信体系架构中扮演的重要角色。HA 队列是 RabbitMQ 团队创建的一项增强功能（未包含在 AMQP 规范中），它允许队列在多个服务器上拥有冗余副本。

HA 队列需要 RabbitMQ 集群环境，可以通过以下两种方式之一进行设置：使用 AMQP 或使用基于 Web 的管理界面。在第 8 章中，我们将重新审视高可用队列并使用管理接口来定义高可用队列的策略，但现在我们将重点讨论使用 AMQP。

在以下示例中，你将使用 Queue.Declare AMQP 命令中的传入参数创建一个新队列，该队列跨越 RabbitMQ 集群中的每个节点。该代码位于“4.1.6 HA 队列声明”笔记文件中。

```
import rabbitpy

connection = rabbitpy.Connection()
try:
    with connection.channel() as channel:
        queue = rabbitpy.Queue(channel,
                               'my-ha-queue',
                               arguments={'x-ha-policy': 'all'})
```

作为 guest 连接到本地主机上的 RabbitMQ

创建 Queue 对象的新实例，传递 HA 策略

打开一个信道进行通信

```
        if queue.declare():                                          ← 声明队列
            print('Queue declared')
except rabbitpy.exceptions.RemoteClosedChannelException as error:    ← 捕获错误引发的任何异常
    print('Queue declare failed: %s' % error)
```

当发布消息到设置为高可用的队列中时，该消息会被发送到集群中的每台服务器（见图 4.7），该集群管理着 HA 队列。一旦消息在集群中的任何节点都完成消费，那么消息的所有副本将立即从其他节点中删除。

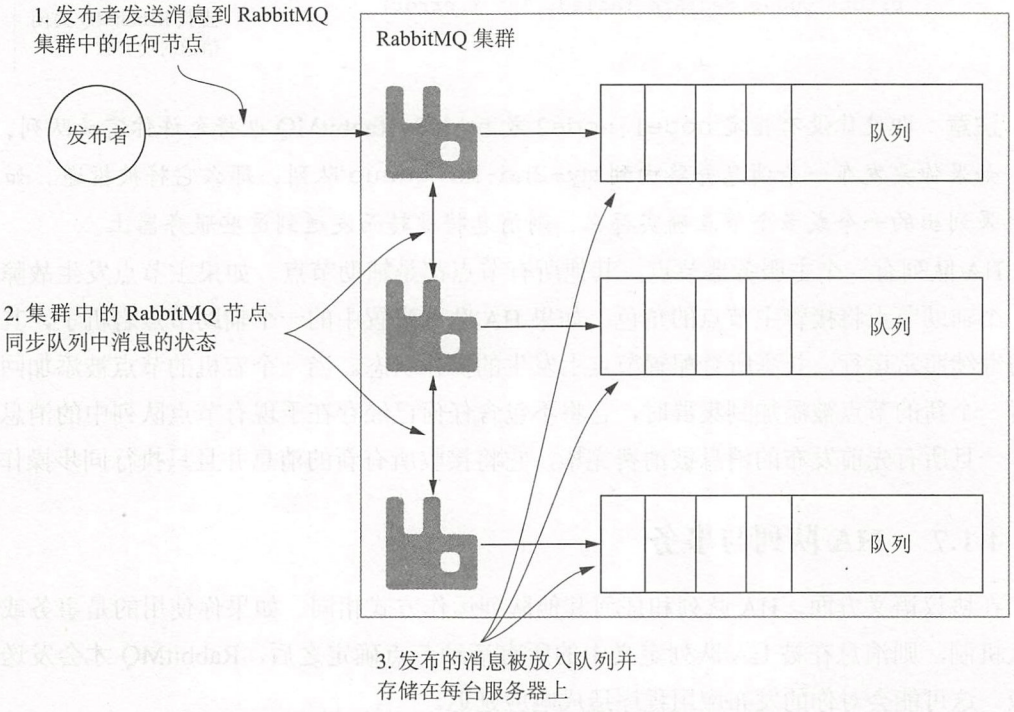


图 4.7　发布到 HA 队列中的消息存储在为其配置的每台服务器上

HA 队列可以跨越集群中的每台服务器，或者仅使用一批独立节点。要指定一批独立节点，不是将 x-ha-policy 参数设置为 all，而是将 x-ha-policy 设置为 nodes，然后再传入另一个参数 x-ha-nodes，该参数包含一个应该配置队列的节点列表。以下示例位于“4.1.6 选择性 HA 队列声明”笔记文件中。

```
import rabbitpy

connection = rabbitpy.Connection()                    ← 连接到 RabbitMQ
try:
    with connection.channel() as channel:             ← 打开一个信道进行通信
```

```
arguments = {'x-ha-policy': 'nodes',                    <-- 指定队列应使用的 HA 策略
             'x-ha-nodes': ['rabbit@node1',
                            'rabbit@node2',
                            'rabbit@node3']}
queue = rabbitpy.Queue(channel,                         <-- 创建Queue对象的新实例，传入 HA 策略和节点列表
                       arguments=arguments)
if queue.declare():                                     <-- 声明队列
    print('Queue declared')
except rabbitpy.exceptions.RemoteClosedChannelException as error:   <-- 捕获 RabbitMQ 关闭信道时的异常
    print('Queue declare failed: %s' % error)
```

注意 即使你没有指定 node1、node2 或 node3，RabbitMQ 也将允许你定义队列，如果你要发布一条消息并路由到 my-2nd-ha-queue 队列，那么它将被投递。如果列出的一个或多个节点确实存在，则消息将被转而发送到这些服务器上。

HA 队列有一个主服务器节点，其他所有节点都是辅助节点。如果主节点发生故障，其中一个辅助节点将接管主节点的角色。如果 HA 队列配置中的一个辅助节点宕机了，其他节点将继续照常运行，共享所有配置节点上发生的操作状态。当一个宕机的节点被添加回来，或者一个新的节点被添加到集群时，它将不包含任何已经存在于现有节点队列中的消息。相反，一旦所有先前发布的消息被消费完毕，它将接收所有新的消息并且只执行同步操作。

4.1.7 HA 队列与事务

在协议语义方面，HA 队列和任何其他队列运作方式相同。如果你使用的是事务或投递确认机制，则消息在被 HA 队列定义中的所有活动节点确定之后，RabbitMQ 才会发送成功响应。这可能会对你的发布应用程序造成响应延迟。

4.1.8 通过设置 delivery-mode 为 2 将消息持久化到磁盘

你已经了解到如何使用备用交换器来处理无法路由的消息。现在是时候为它们添加另一种级别的可靠投递机制了。如果 RabbitMQ 代理服务器在消费消息之前因某种原因发生宕机，那么消息将永远丢失，除非你在发布消息时告诉 RabbitMQ 你希望在消息处理过程中将它们保存在磁盘上。

正如你在第 3 章中了解到的，消息属性是 AMQP Basic.Properties 定义的一部分，而 delivery-mode 是 AMQP 中的一个消息属性。如果将一个消息的 delivery-mode 设置为 1（也就是默认值），RabbitMQ 会被告知不需要将消息存储到磁盘，而消息会一直保存在内

存中。因此，如果对 RabbitMQ 进行重启，那么当 RabbitMQ 启动成功并运行时，非持久化消息将不可用。

另一方面，如果将 delivery-mode 设置为 2，RabbitMQ 将确保消息存储到磁盘。这通常被称为消息持久化（message persistence），将消息存储到磁盘可确保如果 RabbitMQ 代理服务器因任何原因进行重启之后，消息仍然在队列中。

注意　为了使消息在 RabbitMQ 代理服务器重启之后仍然存在，除了将 delivery-mode 设置为 2，你的队列在创建时必须声明为持久性（durable）。第 5 章将详细介绍持久性队列。

对于没有足够 I/O 处理能力的服务器来说，消息持久化可能会导致严重的性能问题。与高访问量 Web 应用程序中的数据库服务器类似，高访问量的 RabbitMQ 实例必须经常操作磁盘以持久化消息。

对于大多数动态 Web 应用程序来说，OLTP 数据库的读写比率中读操作占有绝大部分（见图 4.8）。对于维基百科等内容网站尤其如此。在这些场景中，有数以百万计的文章，其中许多正处于创建或更新中，但大多数用户正在阅读内容，而不是写内容。

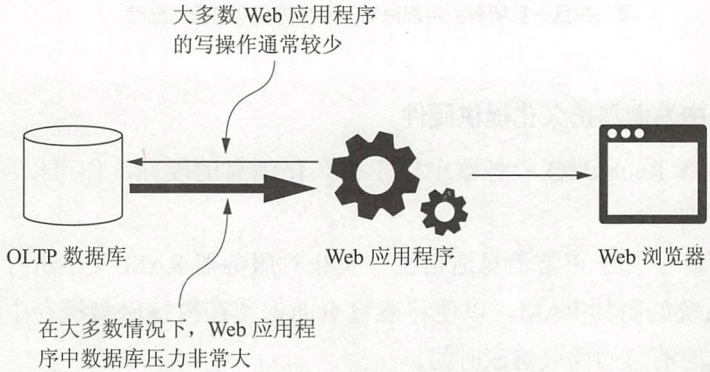


图 4.8　虽然情况并非总是如此，相较在生成网页时将内容写入到数据库中，大多数 Web 应用程序更多的是从数据库中读取数据

当 RabbitMQ 持久化消息时，可能会有相当大的写入偏差（见图 4.9）。在高吞吐量的消息通信环境中，RabbitMQ 将持久化消息写入磁盘，并通过引用追踪它们直到它们不存在于任何队列中为止。一旦消息的所有引用消失，RabbitMQ 将从磁盘中删除消息。在进行高速写入时，经常会发生由于硬件配置不足而导致的性能问题，因为在大多数情况下，磁盘的写缓存要比读缓存小得多。在大多数操作系统中，内核将使用空余 RAM 来缓冲从磁盘读取的页面，而将缓存写入磁盘的组件只有磁盘控制器和磁盘。正因为如此，在使用持久化消

息时正确评估硬件需求非常重要。一个容量不足的服务器执行大量写入操作可能导致整个 RabbitMQ 服务器性能急剧下降。

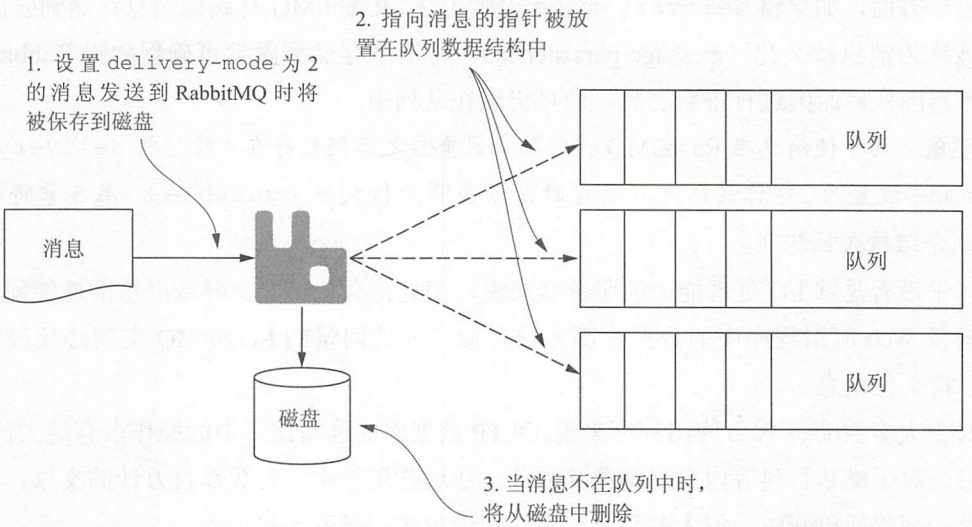


图 4.9　RabbitMQ 一次存储一个持久化消息，并跟踪它存储在所有队列的引用。如果可能的话，避免磁盘读，并且一旦所有引用都消失，消息将会从磁盘上删除

在 RabbitMQ 中为消息持久化提供硬件

为了正确配置 RabbitMQ 服务器中用于持久化消息的硬件，你可以应用与 OLTP 数据库相同的规则。

内存为王：除了基于正常消息通信工作负载对服务器 RAM 大小进行调整之外，还要考虑操作系统的附加 RAM，以便将磁盘页面保留在内核磁盘缓存中。这将改善已经从磁盘加载的消息的读取响应时间。

越多越好：虽然固态硬盘可能正在改变常规模式，但这个概念仍然适用。可用硬盘越多，写入吞吐量就越好。由于系统可以将写入工作负载分散到 RAID 设置的所有磁盘上，因此每个物理设备被阻塞的时间将大大缩短。

找到一个适当大小、带有备用电池的 RAID 卡，并配备大量的读写缓存。这会把所写数据缓存到 RAID 卡中，并允许在写入活动中出现短暂的尖峰，否则这些写入活动将会因为物理设备的限制而被阻塞。

在 I/O 密集型服务器中，通过操作系统在存储设备之间传输数据时，操作系统将阻塞 I/O 操作的进程。当 RabbitMQ 服务器正在尝试执行 I/O 操作（例如将信息保存到磁盘），并且在等待存储设备响应时操作系统内核发生阻塞，那么 RabbitMQ 能做的就只有等待。如果 RabbitMQ 代理服务器经常等待操作系统响应读写请求，消息吞吐量将大大降低（见图 4.10）。

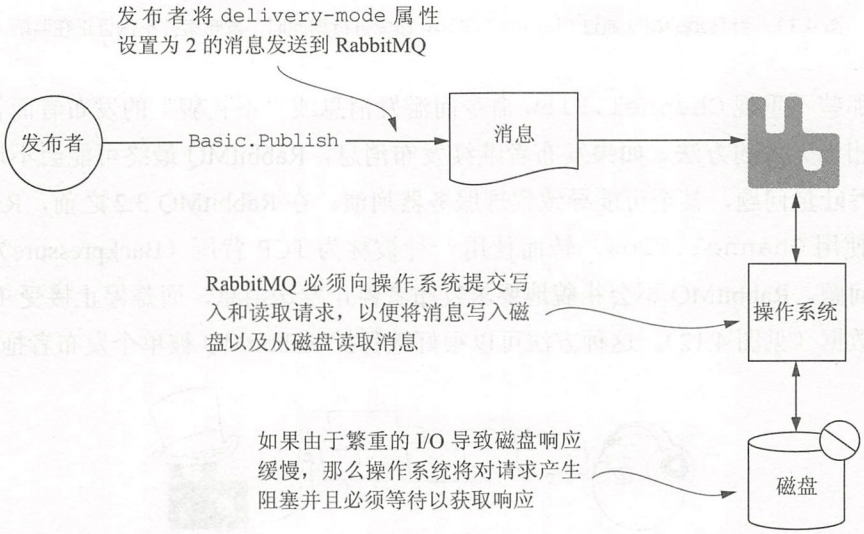


图 4.10　当收到 delivery-mode 属性被设置为 2 的消息时，RabbitMQ 必须将该消息写入磁盘

尽管消息持久化是保障你的消息最终被投递的最重要方式之一，但实现它的代价也是最大的。磁盘性能不佳可能会大大降低 RabbitMQ 的消息发布速度。在极端情况下，硬件配置不当造成的 I/O 延迟可能导致消息丢失。简而言之，如果 RabbitMQ 由于操作系统发生 I/O 阻塞而无法响应发布者或消费者时，那么消息就不能被发布或投递。

4.2　RabbitMQ 回推

在 AMQP 规范中，如果发布者不适合服务器实现就会对发布者创建一些假设。在 RabbitMQ 2.0 版本之前，如果发布者应用程序因为发布消息太快而开始对 RabbitMQ 造成压力，那么 RabbitMQ 将发送 Channel.Flow RPC 方法（见图 4.11）来让你的发布者发生阻塞，即发布者不能发送任何消息直到收到另一条 Channel.Flow 命令为止。

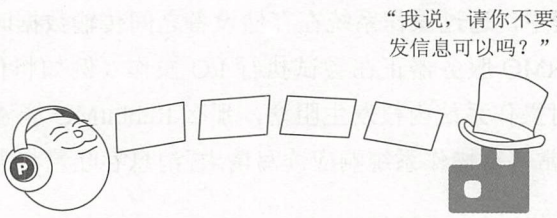


图 4.11　当 RabbitMQ 通过 Channel.Flow 命令进行询问时，发布者并不保证正在监听

对于那些不重视 Channel.Flow 命令而滥发消息或“不礼貌”的发布者而言，这被证明是一种相当无效的方法。如果发布者继续发布消息，RabbitMQ 最终可能会不堪重负，导致性能和吞吐量问题，甚至可能导致代理服务器崩溃。在 RabbitMQ 3.2 之前，RabbitMQ 团队不推荐使用 Channel.Flow，转而使用一种被称为 TCP 背压（Backpressure）的机制来解决这个问题。RabbitMQ 不会礼貌地要求发布者停止发送消息，而是停止接受 TCP 套接字上的低层数据（见图 4.12）。这种方法可以很好地保护 RabbitMQ 被单个发布者拖垮。

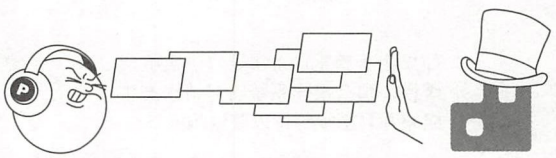


图 4.12　RabbitMQ 应用 TCP 背压来阻止不礼貌的发布者过度发送消息

在内部，RabbitMQ 使用信用的概念来管理回推发布者的时机。在建立新的连接时，连接将被分配一个预定数量的可用信用值。然后，当 RabbitMQ 接收每个 RPC 命令时，将扣除一个点的信用值。一旦 RPC 请求在内部完成处理，连接就会返还被扣除的信用值。连接的信用值余额由 RabbitMQ 评估，以确定它是否应该从连接的套接字读取数据。如果一个连接的信用值不足，它将被跳过直到它有足够的信用值为止。

从 RabbitMQ 3.2 开始，RabbitMQ 团队扩展了 AMQP 规范，添加了在达到连接信用阈值时发送通知的机制，用于通知客户端其连接已被阻塞。Connection.Blocked 和 Connection.Unblocked 是可以随时发送的异步方法，以便在 RabbitMQ 对客户端进行阻塞或取消阻塞时通知到客户端。大多数主要客户端库都实现了这个功能，你应该检查正在使用的特定客户端库，看看你的应用程序应该如何确定连接状态。在下一节中，你将看到如何使用 rabbitpy 执行此检查，以及如何在 RabbitMQ 3.2 之前的版本中使用管理 API 来检查连接的信道是否被阻塞。

注意　最终，TCP 背压和连接阻塞不是每天都应该遇到的问题，它们可以是对 RabbitMQ 服务器硬件大小不是很合理的一种提示。如果你发现这是一个问题，那么是时候评估你的扩容策略，并且可能实现第 8 章中介绍的一些概念。

4.2.1　使用 rabbitpy 来检测连接状态

无论你是否使用支持 `Connection.Blocked` 通知的 RabbitMQ 版本，rabbitpy 库都会将此功能封装到一个易于使用的 API 中。当连接到支持 `Connection.Blocked` 通知的 RabbitMQ 版本时，rabbitpy 将收到通知，并将设置一个内部标志用于指出连接被阻塞。

当你使用“4.2.1 连接阻塞”笔记文件中的以下示例时，输出应该是连接未被阻塞。

```
import rabbitpy

connection = rabbitpy.Connection()
print('Connection is Blocked? %s' % connection.blocked)
```

连接到 RabbitMQ

检查客户端是否被阻塞

4.2.2　使用管理 API 管理连接状态

如果你使用的是 3.2 版本之前的 RabbitMQ，则你的应用程序可以使用基于 Web 的管理 API 来查询连接的状态。这样做相当简单，但如果使用频率过高，可能会导致 RabbitMQ 服务器上不必要的负载压力。根据你的集群大小以及你所拥有的队列数量，该 API 请求可能需要几秒钟才能返回。

管理 API 提供 RESTful URL 端点用于查询连接、信道、队列以及 RabbitMQ 中任何其他向外暴露对象的状态。在管理 API 中，阻塞状态适用于连接中的信道，而不是连接本身。查询信道状态时可以获取多个字段，包括 `name`、`node`、`connection_details`、`consumer_count` 和 `client_flow_blocked` 等。其中 `client_flow_blocked` 标志指示 RabbitMQ 是否将 TCP 背压应用于连接。

要获取信道的状态，你必须先为其创建合适的名称。信道名称基于连接名称和它的信道 ID。要创建连接名称，你需要以下内容：

- 本地主机 IP 地址和对外 TCP 端口。
- 远程主机 IP 地址和 TCP 端口。

格式是“`LOCAL_ADDR:PORT->REMOTE_ADDDR:PORT`”。信道的命名格式在此基础上进行扩展，表现为“`LOCAL_ADDR:PORT->REMOTE_ADDDR:PORT (CHANNEL_ID)`”。

用于查询 RabbitMQ 管理 API 以获取信道状态的 API 端点为 http://host:port/api /channels/

[CHANNEL_NAME]。查询时，管理 API 将以 JSON 序列化对象的格式返回结果。以下是使用 API 查询信道状态所返回的简略示例：

```
{
    "connection_details": {…},
    "publishes": […],
    "message_stats": {…},
    "consumer_details": [],
    "transactional": false,
    "confirm": false,
    "consumer_count": 0,
    "messages_unacknowledged": 0,
    "messages_unconfirmed": 0,
    "messages_uncommitted": 0,
    "acks_uncommitted": 0,
    "prefetch_count": 0,
    "client_flow_blocked": false,
    "node": "rabbit@localhost",
    "name": "127.0.0.1:45250 -> 127.0.0.1:5672 (1)",
    "number": 1,
    "user": "guest",
    "vhost": "guest"
}
```

除了 channel_flow_blocked 字段之外，管理 API 还会返回有关该信道的速率和状态信息。

4.3 小结

创建应用程序体系架构的主要步骤之一是定义发布者的角色和行为。你应该问自己以下问题：

- 发布者是否要求将消息持久化到磁盘？
- 应用程序的各个组件需要什么样的保障机制以确保发布的消息都会被接收？
- 如果应用程序被 TCP 背压阻塞，或者在将消息发布到 RabbitMQ 时连接被阻塞，我的环境中将会发生什么？
- 我的消息有多重要？ 我可以牺牲消息的可靠投递来实现更高的消息吞吐量吗？

通过问自己这些问题，你将创建一个刚刚好的应用程序架构。RabbitMQ 提供了大量的灵活性，在某些情况下可能太多了。但是，通过利用定制功能的优势，你就可以在性能和高可靠性之间进行权衡，并决定适合消息的元数据级别。最好由你自己来决定使用哪些属性和机制来实现可靠消息投递，而 RabbitMQ 将为你的选择提供坚实的基础。

第 5 章　消费消息，避免拉取

本章概要：

- 消费消息
- 优化消费者吞吐量
- 消费者和队列的独占性
- 为消费者指定服务质量

在上一章中我们深入到了消息发布者的世界，现在是讨论对发布者发送的消息进行消费的时候了。消费者应用程序可以是专用应用程序，其唯一目的就是接收消息并对消息进行操作，或者说接收消息可能只是大型应用程序的一小部分。例如，如果你正在使用 RabbitMQ 实现 RPC 模式，发布 RPC 请求的应用程序也正在消费 RPC 回复（见图 5.1）。

有了这么多可以在你的应用程序中实现消息通信的模式，唯一合适的是 RabbitMQ 提供了各种各样的设置用于在性能和可靠消息通信之间找到适当的平衡。决定你的应用程序如何消费消息是找到这种平衡的第一步，这一步从一个简单的选择开始：拉取（get）消息还是消费（consume）消息？ 在这一章中，你将学习：

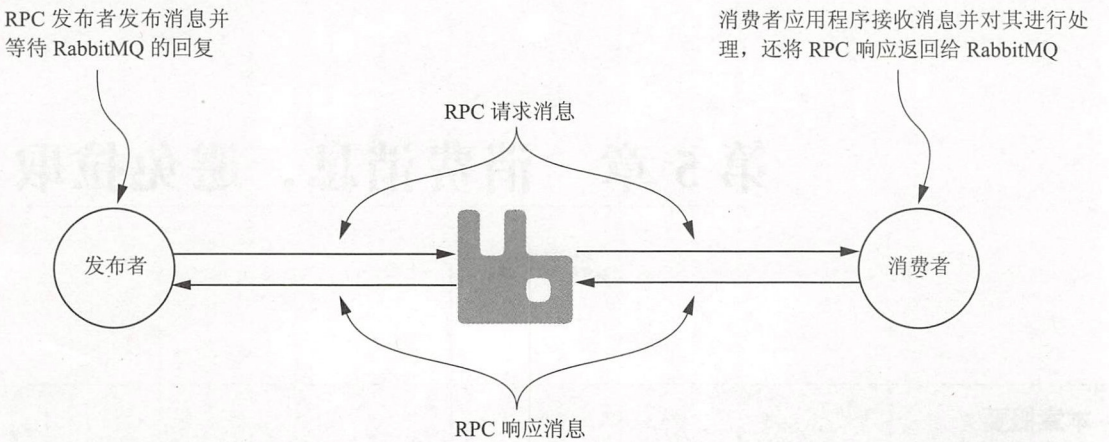


图 5.1　一个 RPC 发布者向 RabbitMQ 发布消息，并作为消费者等待来自 RPC 消费者的回复

- 为什么你应该避免拉取消息，而应该倾向于消费消息。
- 如何平衡消息投递的可靠性与性能。
- 如何使用RabbitMQ队列级别的设置来实现自动删除队列、限制消息的生存时间等功能。

5.1　对比 Basic.Get 和 Basic.Consume

RabbitMQ 实现了两个不同的 AMQP RPC 命令来获取队列中的消息：Basic.Get 和 Basic.Consume。正如本章标题所示，Basic.Get 不是从服务器获取消息的理想方法。使用最简单的说法，Basic.Get 是一个轮询模型，而 Basic.Consume 是一个推送模型。

5.1.1　Basic.Get

当你的应用程序使用 Basic.Get 请求来获取消息时，每次它想要接收消息就必须发送一个新的请求，即使队列中存在多个消息。当发出一个 Basic.Get，如果你想要获取消息的队列中有一条消息正处于等待处理状态，RabbitMQ 就会回应一个 Basic.GetOk RPC 响应（见图 5.2）。

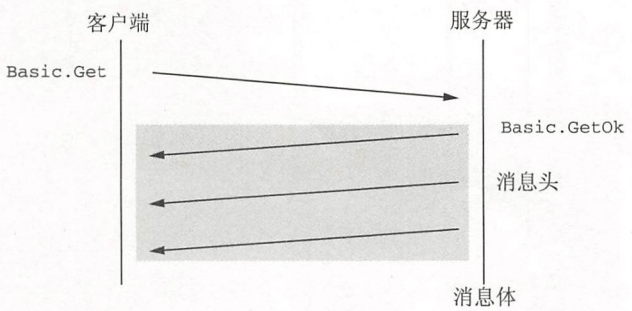


图 5.2　如果在发出 Basic.Get RPC 请求时有一条消息可用，RabbitMQ 将返回 Basic.GetOk 以及消息本身

如果队列中没有待处理的消息，它将回复 Basic.GetEmpty，表示队列中没有消息（见图 5.3）。

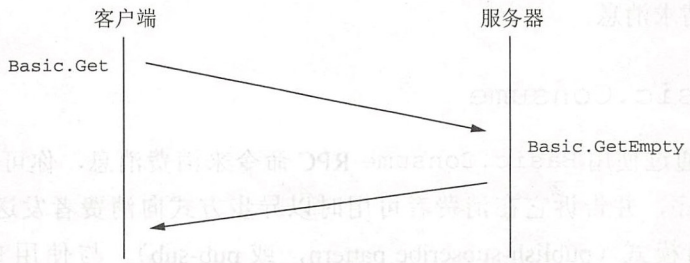
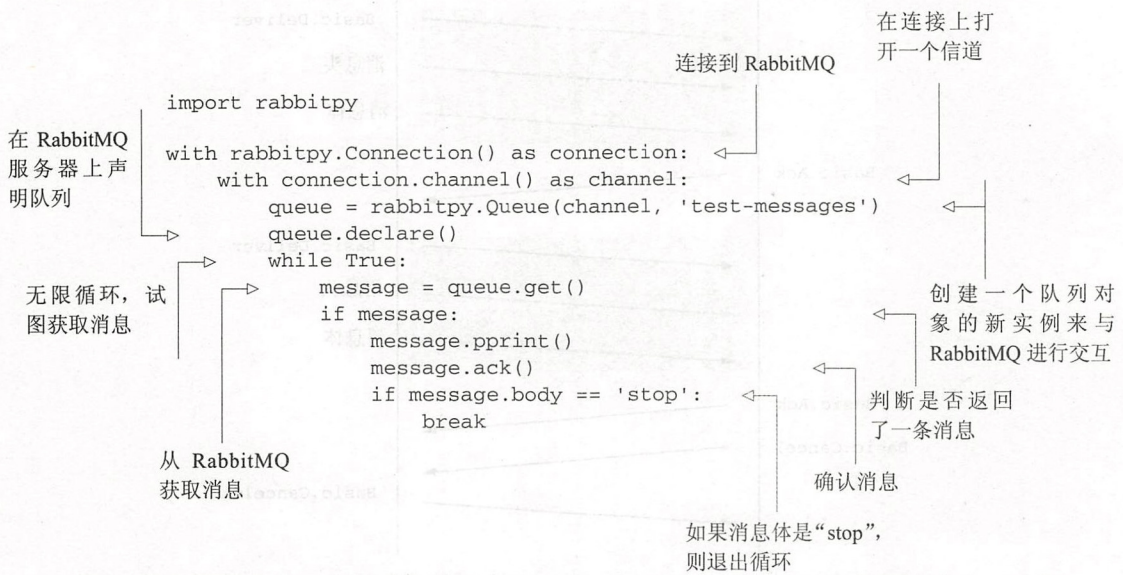


图 5.3　如果发出 Basic.Get 请求时没有消息可用，则 RabbitMQ 使用 Basic.GetEmpty 进行响应

当使用 Basic.Get 时，你的应用程序应评估来自 RabbitMQ 的 RPC 响应以确定是否收到了消息。对于大多数从 RabbitMQ 接收消息并长时间运行的流程来说，这并不是一种接收和处理消息的有效方式。

参考“5.1.1 Basic.Get 示例”笔记文件中的代码。连接到 RabbitMQ 并打开信道后，它会使用无限循环向 RabbitMQ 请求消息。




```
import rabbitpy

with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        queue = rabbitpy.Queue(channel, 'test-messages')
        queue.declare()
        while True:
            message = queue.get()
            if message:
                message.pprint()
                message.ack()
                if message.body == 'stop':
                    break
```

虽然这是与 RabbitMQ 进行交互以获取消息的最简单方法，但在大多数情况下，性能顶多算是可以接受的。在简单的消息速度测试中，使用 Basic.Consume 至少是使用 Basic.Get 的两倍。速度不同的最明显原因是使用 Basic.Get 会导致每条消息都会产生与 RabbitMQ 同步通信的开销，这一过程由发送请求帧的客户端应用程序和发送应答的 RabbitMQ 组成。避免使用 Basic.Get 的一个潜在的不太明显的原因是它会影响吞吐量，

由于 Basic.Get 的临时性，RabbitMQ 不能以任何方式优化投递过程，因为它永远不知道应用程序何时会请求消息。

5.1.2 Basic.Consume

相比之下，通过使用 Basic.Consume RPC 命令来消费消息，你可以使用 RabbitMQ 注册你的应用程序，并告诉它在消费者可用时以异步方式向消费者发送消息。这通常被称为发布—订阅模式（publish-subscribe pattern，或 pub-sub）。与使用 Basic.Get 时与 RabbitMQ 创建的同步会话不同，使用 Basic.Consume 消费消息意味着你的应用程序会在消息可用时自动从 RabbitMQ 接收消息，直到客户端发出 Basic.Cancel 为止（见图 5.4）。

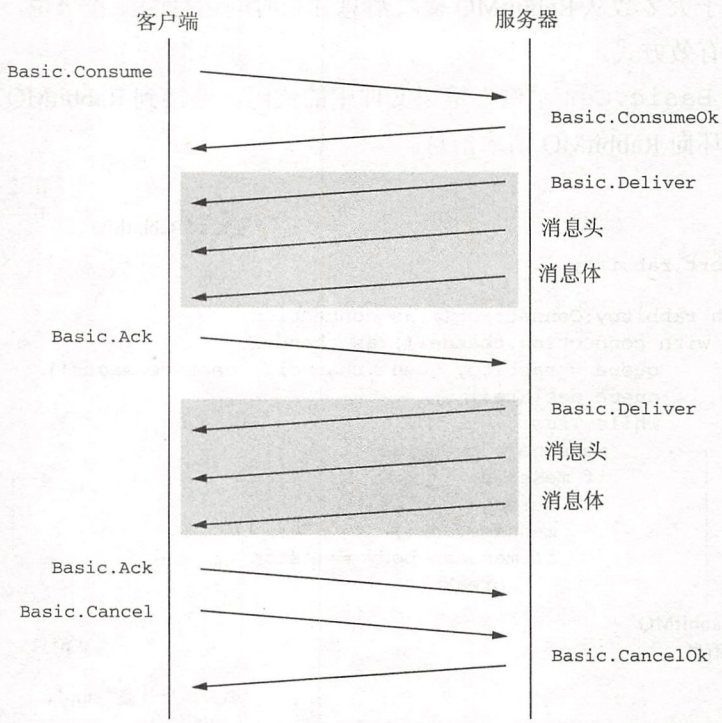


图 5.4 当一个客户端发出一个 Basic.Consume 时，一旦有消息可用时 RabbitMQ 就会进行发送，直到客户端发出一个 Basic.Cancel 为止

当收到消息时，从 RabbitMQ 消费消息在代码上会少一步。如下例所示，当应用程序作为消费者接收到来自 RabbitMQ 的消息时，它不需要评估消息以确定该值是消息还是一个空响应（Basic.GetEmpty）。但是就像 Basic.Get 一样，你的应用程序仍然需要确认消息以便

让 RabbitMQ 知道消息已经被处理。该代码包含在“5.1.2 Basic.Consume 示例”笔记文件中。

```
import rabbitpy

for message in rabbitpy.consume('amqp://guest:guest@localhost:5672/%2f',
                                'test-messages'):
    message.pprint()
    message.ack()
```

遍历测试消息队列中的消息

确认收到消息

注意　你可能已经注意到以上示例中的代码比上一个示例中的代码短。这是因为 rabbitpy 提供了简化方法，封装了连接到 RabbitMQ 和使用信道所需的大部分逻辑。

消费者标签

当你的应用程序发出 Basic.Consume 时会创建一个唯一的字符串，用来标识通过已建立的信道与 RabbitMQ 进行通信的应用程序。这个字符串被称为消费者标签（Consumer Tag），RabbitMQ 每次都会把该字符串与消息一起发送给你的应用程序。

通过发送一个 Basic.Cancel RPC 命令，消费者标签可以用来取消从 RabbitMQ 获取消息。如果你的应用程序同时从多个队列中消费消息，这点就非常有用，因为每个收到的消息都在它的方法帧中包含该消息所投递的目标消费者标签。如果你的应用程序需要对从不同队列接收到的消息执行不同的操作，则可以使用 Basic.Consume 请求中的消费者标签来确定如何处理消息。但是，在大多数情况下，客户端库已经对消费者标签做了封装，所以你不必担心它。

在“5.1.2 消费者与 Stop 消息”笔记文件中，以下消费者代码将监听消息，直至收到仅包含“Stop”消息体的消息为止。

```
import rabbitpy

with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        for message in rabbitpy.Queue(channel, 'test-messages'):
            message.pprint()
            message.ack()
            if message.body == 'stop':
                break
```

连接到 RabbitMQ

在连接上打开一个信道

作为消费者遍历队列中的消息

打印消息属性

确认消息

判断消息体，当发现是“stop”时停止消费

消费者启动后，可以使用“5.1.2 消息发布者”笔记文件中的代码在新的浏览器页面中发布消息：

```
import rabbitpy

for iteration in range(10):
    rabbitpy.publish('amqp://guest:guest@localhost:5672/%2f',
                     '', 'test-messages', 'go')
rabbitpy.publish('amqp://guest:guest@localhost:5672/%2f',
                 '', 'test-messages', 'stop')
```

循环 10 次

向 RabbitMQ 发布相同的消息

向 RabbitMQ 发布 stop 消息

当你运行发布者，一旦在退出 Queue.consume_messages 循环时收到 stop 消息，“5.1 消费者与 Stop 消息示例”笔记文件中的代码将停止运行。当退出循环时，rabbitpy 库帮我们做了一些事情。首先，rabbitpy 库发送一个 Basic.Cancel 命令给 RabbitMQ。如果 RabbitMQ 发送到客户端的某个消息没有得到处理，那么一旦接收到 Basic.CancelOk RPC 响应，rabbitpy 将发送一个否定确认命令（Basic.Nack），并指示 RabbitMQ 重新发送消息。

在同步 Basic.Get 和异步 Basic.Consume 之间进行选择是在编写消费者应用程序时需要做的几个决策之一。与发布消息时涉及的权衡一样，你为应用程序所做的选择可能会直接影响消息的可靠投递和性能。

5.2 优化消费者性能

当发布消息时，对消息的消费在吞吐量与可靠投递之间存在一种平衡。如图 5.5 所示，有几个选项可用于加速 RabbitMQ 和应用程序之间的消息投递。当发布消息时，RabbitMQ 也提供了一些选项，在弱化消息投递保证的同时提高消息投递的吞吐量。

在本节中，你将了解如何通过切换消息确认的需求以优化 RabbitMQ 消息投递的消费者吞吐量、如何调整 RabbitMQ 的消息预分配阈值以及如何评估使用消费者事务所产生的影响。

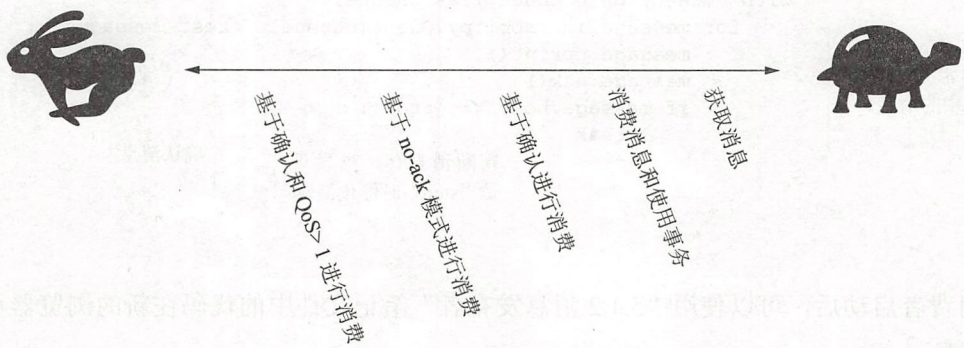
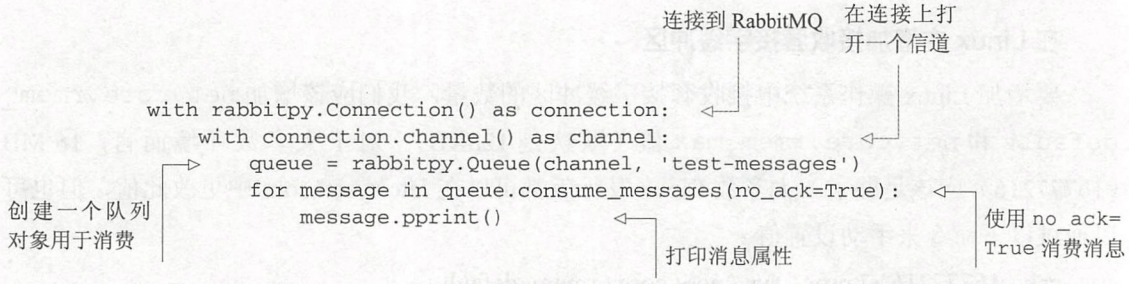


图 5.5 消费者性能优化维度

5.2.1　使用 no-ack 模式实现更快的吞吐量

在消费消息时，应用程序将自己注册到 RabbitMQ，并要求消息在可用时进行投递。你的应用程序发送一个 Basic.Consume RPC 请求，与该请求一起发送的还有一个 no-ack 标志。当这个标志被启用时，它会告诉 RabbitMQ 你的消费者在接收到消息时不会进行确认，RabbitMQ 只管尽快发送它们。

在“5.2.1No-Ack 消费者”笔记文件中的以下示例演示了如何消费消息而不对它们进行确认。通过将 Queue.consumer 方法的参数设置为 True，rabbitpy 将使用 no_ack=True 发送 Basic.Consume RPC 请求。



使用 no_ack=True 消费消息是让 RabbitMQ 将消息投递给消费者的最快方式，但这也是发送消息最不可靠的方式。要理解其中的原因，重要的是要分析消费者应用程序在收到消息之前必须经过的每一步（见图 5.6）。

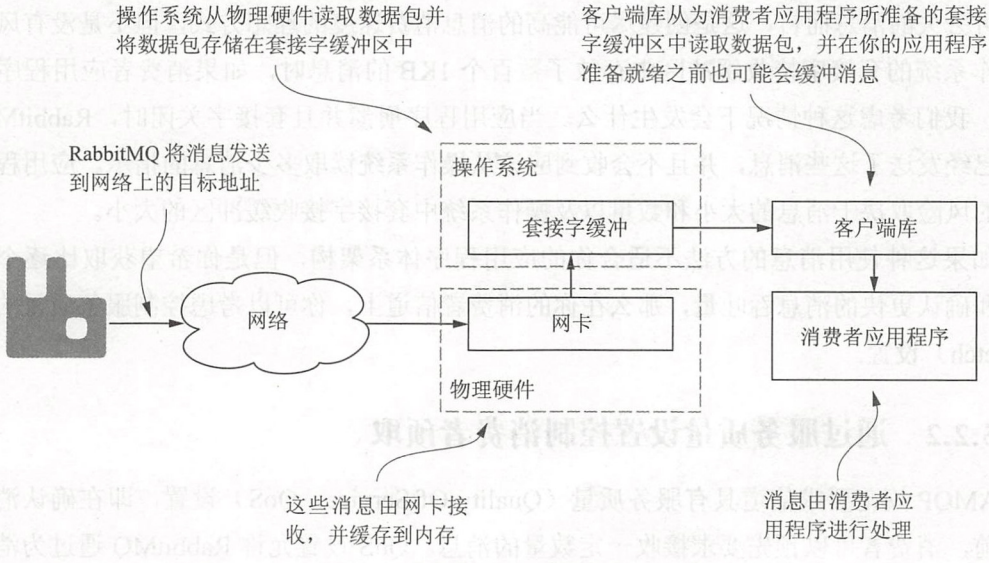


图 5.6　在消费者应用程序之前有多个数据缓冲区接收消息数据

当 RabbitMQ 通过打开的连接发送消息时，它使用 TCP 套接字连接与客户端进行通信。如果这个连接是打开且可写的，那么 RabbitMQ 假定一切都处于正常工作状态并且成功投递了消息。如果当 RabbitMQ 尝试写入套接字以投递消息时出现了网络问题，操作系统将触发套接字错误从而让 RabbitMQ 知道出现了问题。如果没有发生错误，RabbitMQ 将假定消息投递成功。通过 `Basic.Ack` RPC 响应发送的消息确认是客户端让 RabbitMQ 知道已成功接收消息的一种方法，这也是大多数情况下处理消息的方式。但是如果关闭消息确认，那么当有新的可用消息时，RabbitMQ 将会发送该消息而不用等待。实际上，如果有可用消息，RabbitMQ 将会持续向消费者发送它们直到套接字缓冲区被填满为止。

在 Linux 中增加接收套接字缓冲区

要增加 Linux 操作系统中接收套接字缓冲区的数量，我们应该增加 `net.core.rmem_default` 和 `net.core.rmem_max` 值（默认是 128KB）。对于大多数环境而言，16 MB（16777216）应该足够了。大多数 Linux 发行版都可以在 /etc/sysctl.conf 中更改此值，但也可以通过以下命令来手动设置值：

```
echo 16777216> / proc / sys / net / core / rmem_default
echo 16777216> / proc / sys / net / core / rmem_max
```

因为 RabbitMQ 并没有等待消息的确认，这种消费消息的方法通常能提供最高的吞吐量。对于可丢失的消息而言，这是创建尽可能高的消息消费速度的理想方式，但不是没有风险。当操作系统的套接字接收缓冲区中存放了数百个 1KB 的消息时，如果消费者应用程序发生崩溃，我们考虑这种情况下会发生什么。当应用程序崩溃并且套接字关闭时，RabbitMQ 认为它已经发送了这些消息，并且不会收到应该从操作系统读取多少消息的指示。应用程序所面临的风险取决于消息的大小和数量以及操作系统中套接字接收缓冲区的大小。

如果这种使用消息的方法不适合你的应用程序体系架构，但是你希望获取比逐个消息投递和确认更快的消息吞吐量，那么在你的消费者信道上，你可以考虑控制服务质量的预取（prefetch）设置。

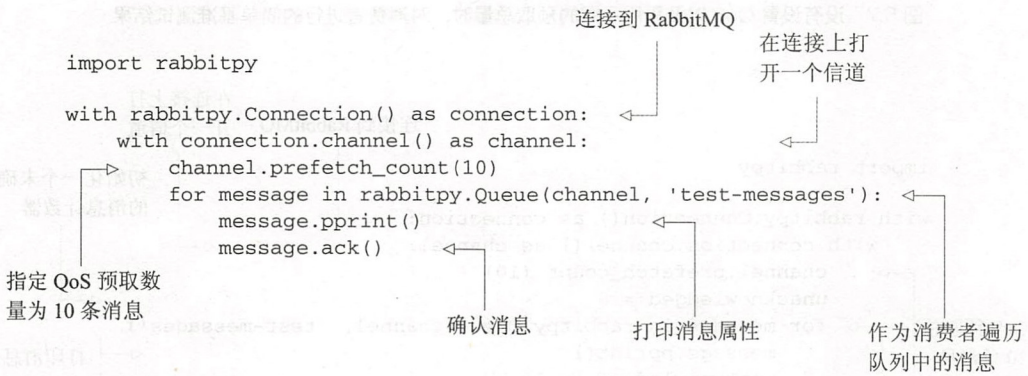
5.2.2 通过服务质量设置控制消费者预取

AMQP 规范要求信道具有服务质量（Quality Of Service，QoS）设置，即在确认消息接收之前，消费者可以预先要求接收一定数量的消息。QoS 设置允许 RabbitMQ 通过为消费者

预先分配一定数量的消息来实现更高效地消息发送。

与被禁用确认（no_ack=True）的消费者不同，如果消费者应用程序在确认消息之前崩溃，则在套接字关闭时，所有预取的消息将返回到队列。

在协议级别，可以在信道上发送 Basic.QoS RPC 请求来指定服务质量。作为这个 RPC 请求的一部分，你可以指定 QoS 设置是针对其发送的信道还是针对连接上打开的所有信道。Basic.QoS RPC 请求可以随时发送，但是如“5.2.2 指定 QoS”笔记文件中的以下代码所示，该请求通常在用户发出 Basic.Consume RPC 请求之前进行发送。



注意　虽然 AMQP 规范要求 Basic.QoS 方法同时设置预取总量和预取大小，但如果设置了 no-ack 选项，预取大小将被忽略。

将你的预取值调整到最佳水平

认识到过度分配预取总量会对消息吞吐量有负面影响也是很重要的。同一队列中的多个用户将以循环（round-robin）方式从 RabbitMQ 接收消息，但是在高访问量消费者应用程序中对预取总量性能进行基准测试非常重要。特定设置的好处可能根据消息组成、消费者行为以及操作系统和语言等其他因素而有所不同。

使用单个消费者对一个简单消息进行基准测试，图 5.7 表明在这种情况下，预取总量为 2,500 是消息速率达到峰值的最佳设置。

一次确认多个消息

使用 QoS 设置的好处之一就是不需要用 Basic.Ack RPC 响应来确认收到的每条消息。相反，Basic.Ack RPC 响应具有一个名为 multiple 的属性，当把它设置为 True 时就能让 RabbitMQ 知道你的应用程序想要确认所有以前未确认的消息。“5.2.2 多消息确定消费者”笔记文件中的以下示例演示了这一点。

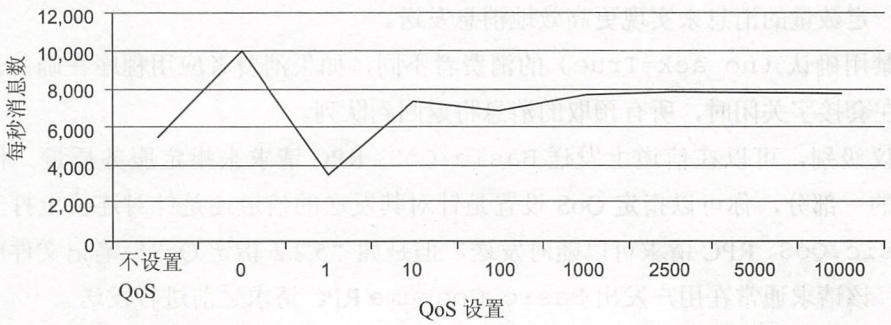
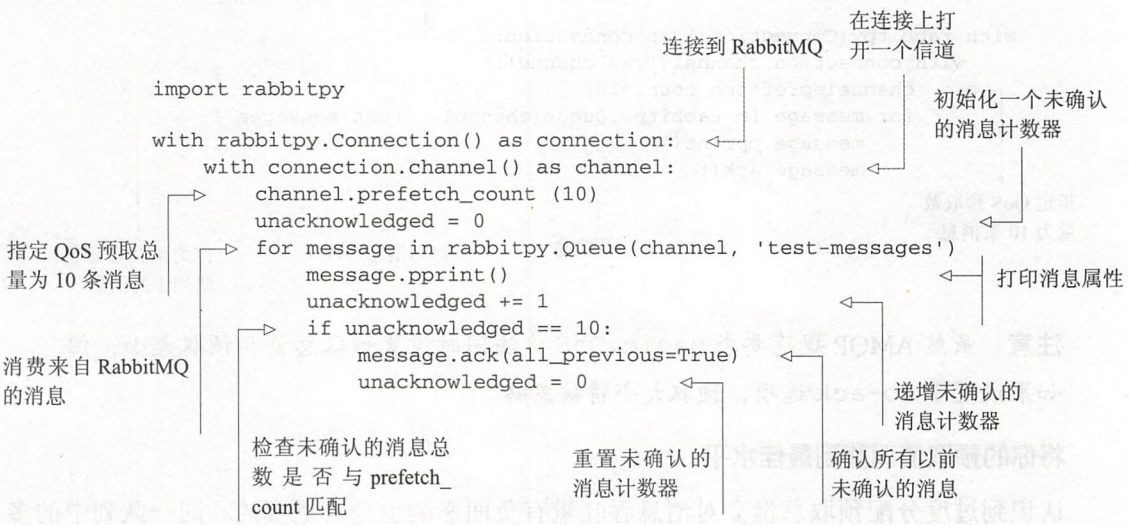


图 5.7　没有设置 QoS 以及采用不同的预取总量时，对消费者进行的简单基准测试结果



同时确认多个消息可以使处理消息所需的网络通信量最小化，从而提高消息吞吐量（见图 5.8）。值得注意的是，这种确认带有某种程度的风险。如果你成功地处理了一些消息，并且你的应用程序在确认它们之前就已经死亡，则所有未确认的消息将返回队列以供其他消费者进行处理。

与发布消息一样，金发姑娘原则也适用于消息消费——你需要找到可接受风险和最佳性能之间的合适位置。除了 QoS 之外，你还应该考虑使用事务来改善应用程序的消息投递可靠性。这些基准测试的源代码可在本书的网站 http://www.manning.com/roy 上找到。

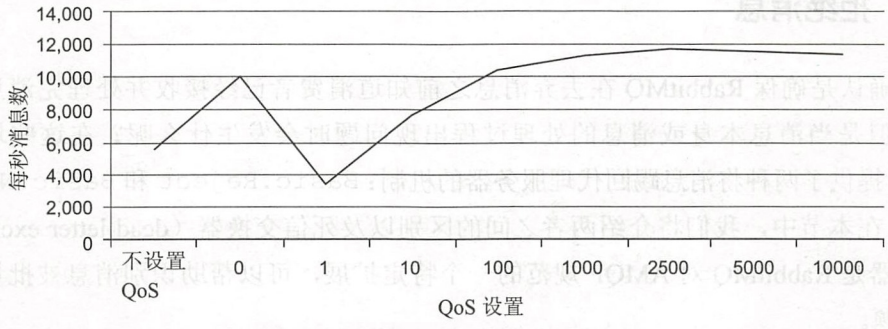


图 5.8　同时确认多个消息提高了吞吐量

5.2.3　消费者使用事务

就像将消息发布到 RabbitMQ 时一样，事务处理允许消费者应用程序提交和回滚批量操作。事务（AMQP TX 类）可能会对消息吞吐量产生负面影响，但有一个例外。如果你不使用 QoS 设置，那么在使用事务来批量确认消息时，实际上可能会看到略微的性能提升（见图 5.9）。

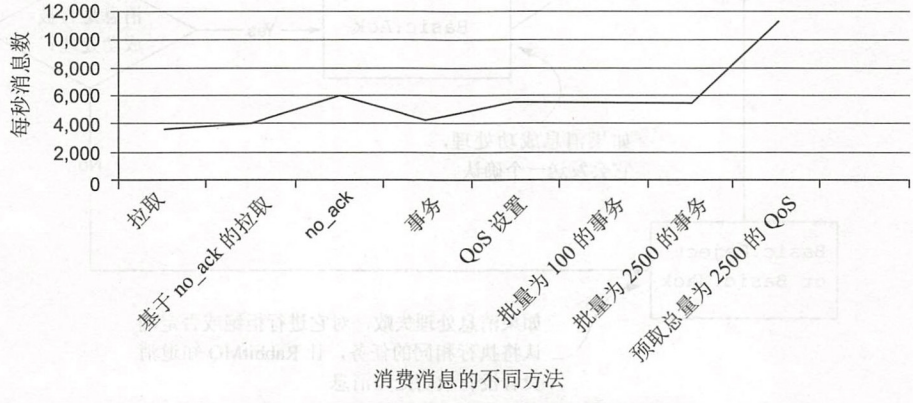


图 5.9　使用或不使用事务时的消息消费速度对比

与使用具体的 QoS 设置一样，你应该将消费者应用程序性能基准测试作为评估的一部分，以确定事务是否应该在消费者应用程序中发挥作用。无论是使用它们进行批量消息确认还是确保在消费消息时可以回滚 RPC 响应，了解事务对性能的真实影响将帮助你在消息可靠投递和消息吞吐量之间找到适当的平衡。

注意　事务不适用于已禁用确认的消费者。

5.3 拒绝消息

消息确认是确保 RabbitMQ 在丢弃消息之前知道消费者已经接收并处理完消息的一种好方法，但是当消息本身或消息的处理过程出现问题时会发生什么呢？在这些场景下，RabbitMQ 提供了两种将消息踢回代理服务器的机制：`Basic.Reject` 和 `Basic.Nack`（见图 5.10）。在本节中，我们将介绍两者之间的区别以及死信交换器（dead-letter exchange），死信交换器是 RabbitMQ 对 AMQP 规范的一个特定扩展，可以帮助识别消息被批量拒绝时的系统问题。

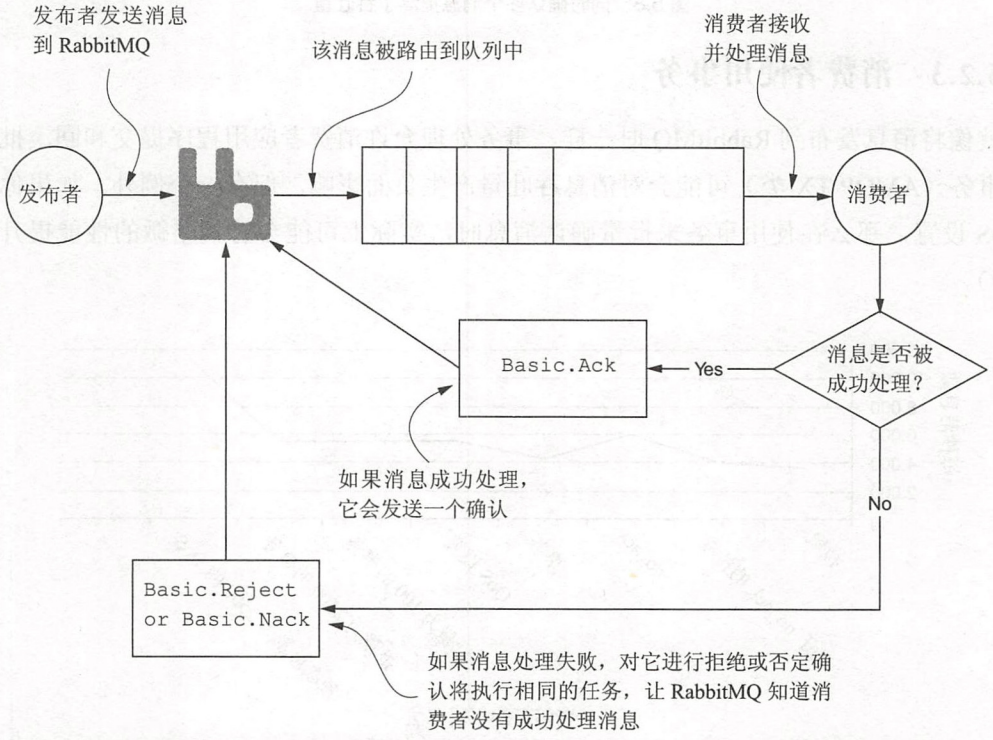


图 5.10 消费者可以对消息进行确认、拒绝或否定确认。`Basic.Nack` 允许一次拒绝多个消息，而 `Basic.Reject` 一次只允许拒绝一个消息

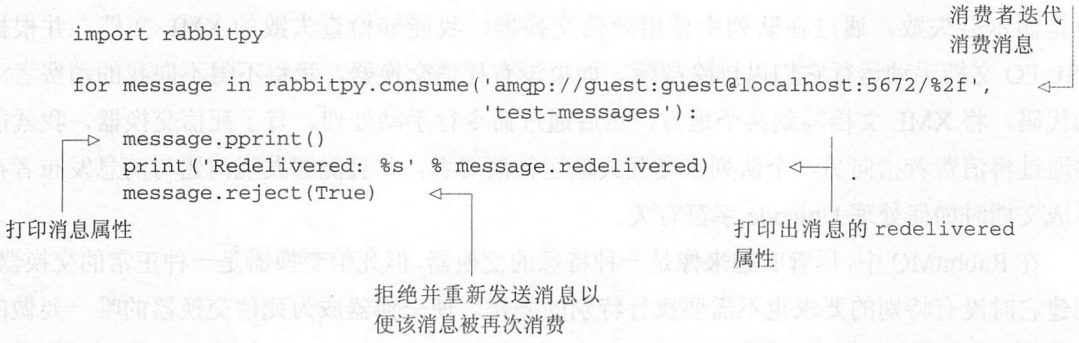
5.3.1 `Basic.Reject`

`Basic.Reject` 是一个 AMQP 指定的 RPC 响应，用于通知代理服务器无法对所投递的消息进行处理。像 `Basic.Ack` 一样，它携带由 RabbitMQ 创建的投递标签，用于唯一标识消费者与 RabbitMQ 进行通信的信道上的消息。当消费者拒绝消息时，你可以指

示 RabbitMQ 丢弃消息或使用 requeue 标志重新发送消息。当启用 requeue 标志时，RabbitMQ 将把消息放回到队列中并再次处理。

我经常使用此功能编写与数据库或远程 API 等其他服务进行通信的消费者应用程序。我并不是在消费者中编写处理远程异常（如一个断开的数据库光标或无法与远程 API 进行交互）的重试逻辑，我只是捕获异常，并将 requeue 设置为 True 以拒绝消息。这使我能够简化消费者中的代码路径，并且在与 Graphite 等统计程序结合使用时，可以通过观察消息重新发送的速度来了解异常行为的趋势。

以下来自“5.3.1 消息拒绝”笔记文件的示例演示了消息是如何进行重新发送的。该消息中设置了 redelivered 标志，用于通知消息的下一个消费者它以前已经被投递过。我已经使用这个功能来执行“双击和出局（two-strikes and you’re out”）”策略。格式错误的消息可能会对消费者造成严重破坏，但是如果你不确定是消息本身还是消费者的其他原因引发了错误，那么检查 redelivered 标志是一个好方法，可以帮你在碰到问题时决定是否应该拒绝那些要重新发送或丢弃的消息。



像 Basic.Ack 一样，如果在消息投递之后没有启用 no-ack 标识，使用 Basic.Reject 会释放对消息的持有。尽管你可以使用 Basic.Ack 一次性确认接收或处理多个消息，但不能使用 Basic.Reject 同时拒绝多个消息——要达到这个效果就需要使用 Basic.Nack。

5.3.2 Basic.Nack

Basic.Reject 允许拒绝单个消息，但是如果你正在使用一个可以利用 Basic.Ack 多消息模式的工作流程，则可能希望在拒绝消息时能够使用类似的功能。不幸的是，AMQP 规范不提供这种行为。RabbitMQ 团队认为这是规范中的一个缺点，并且实现了一种新的 RPC 响应方法，取名为 Basic.Nack。Basic.Nack 是“negative acknowledgment（否定确认）”的缩写，Basic.Nack 和 Basic.Reject 响应方法的相似性在首次接触时可能比较容易混

淆。总而言之，Basic.Nack 方法实现与 Basic.Reject 响应方法相同的行为，但添加了所缺的多消息参数来对 Basic.Ack 多消息处理行为进行补充。

警告 与 RabbitMQ 对 AMQP 协议实现的任何专有扩展一样，其他 AMQP 代理服务器（如 QPID 或 ActiveMQ）中不保证存在 Basic.Nack。另外，没有 RabbitMQ 专用协议扩展的通用 AMQP 客户端将不支持它。

5.3.3 死信交换器

RabbitMQ 的死信交换器（Dead-Letter eXchange，DLX）功能是对 AMQP 规范的扩展，是一种可以拒绝已投递消息的可选行为。在尝试诊断为何消费特定消息会出现问题时，这个功能非常有用。

例如，我编写的一种消费者应用程序消费基于 XML 的消息，并使用称为 XSL:FO 的标准标记语言将其转换为 PDF 文件。通过结合 XSL:FO 文档和消息中的 XML，我能够使用 Apache 的 FOP 应用程序生成一个 PDF 文件，然后进行电子存档。这个流程工作得很好，但是偶尔会失败。通过在队列中使用死信交换器，我能够检查失败的 XML 文档，并根据 XSL:FO 文档手动运行它们以排除故障。如果没有死信交换器，我将不得不向我的消费者添加代码，将 XML 文档写到某个地方，然后通过命令行手动处理。有了死信交换器，我就能够通过将消费者指向另一个队列来交互式地运行消费者，并且能够发现问题与消息发布者在生成文档时如何处理 Unicode 字符有关。

在 RabbitMQ 中，尽管听起来像是一种特殊的交换器，但死信交换器是一种正常的交换器。创建它时没有特别的要求也不需要执行特别的操作。使交换器成为死信交换器的唯一要做的事情是在创建队列时声明该交换器将被用作保存被拒绝的消息。一旦拒绝了一个不重新发送的消息，RabbitMQ 将把消息路由到队列的 x-dead-letter-exchange 参数中指定的交换器（见图 5.11）。

注意 死信交换器与第 4 章讨论的备用交换器不同。过期或被拒绝的消息通过死信交换器进行投递，而备用交换器则路由那些无法由 RabbitMQ 路由的信息。

声明队列时指定死信交换器是相当简单的。只需在创建 rabbitpy Queue 对象时将交换器名称作为 dead_letter_exchange 参数进行传入，或者在发出 Queue.Declare RPC 请求时作为 x-dead-letter-exchange 参数进行传入。自定义参数允许你指定与队列定义一起存储的任意键 / 值对。你将在第 5.4.3 节中了解更多关于它们的内容。以下示例位于

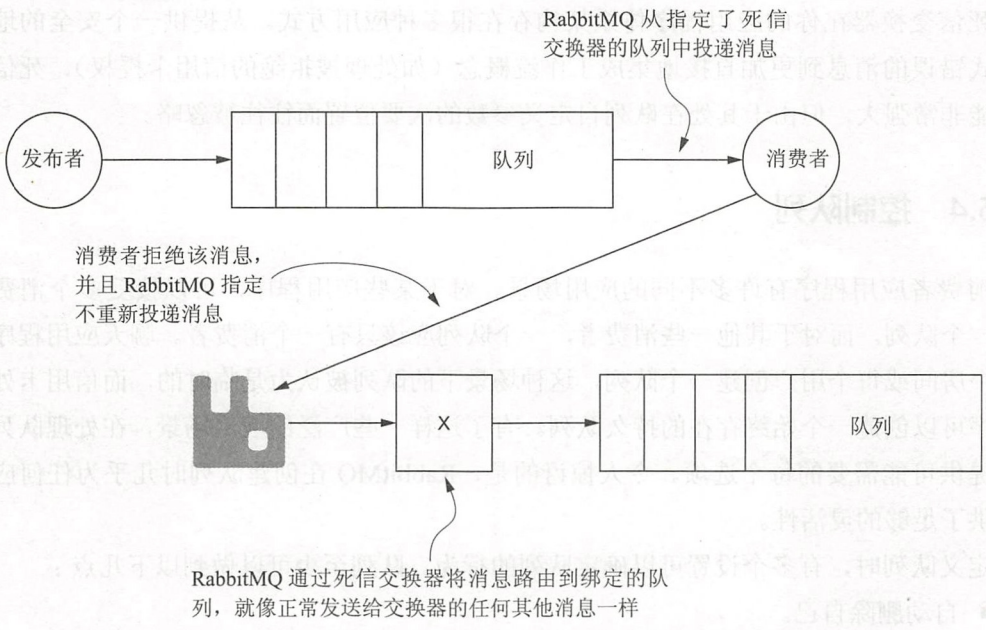
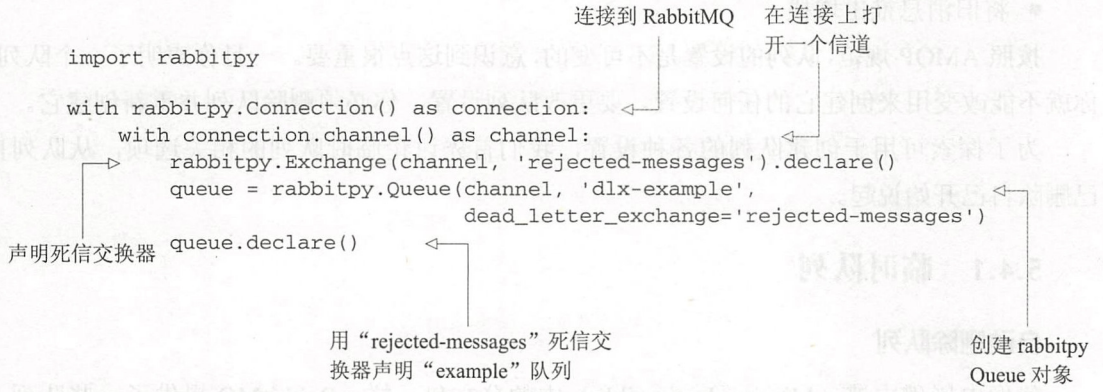


图 5.11　被拒绝的消息可以作为死信消息由另一个交换器进行路由

“5.3.3 指定死信交换器”笔记文件中。



除交换器外，死信功能还允许你使用预先指定的值覆盖路由键。这样可以允许你使用同一个交换器同时处理死信消息和非死信消息，但需要确保死信消息不被投递到相同的队列。设置预定义的路由键需要在声明队列时指定一个额外的参数 `x-dead-letter-routing-key`。

注意　根据 AMQP 标准，RabbitMQ 中的所有队列设置都是不可变的，这意味着在队列被声明后它们不能被修改。为了改变队列的死信交换器，你必须删除并重新声明它。

死信交换器在你的应用程序体系架构存在很多种应用方式。从提供一个安全的地方存储格式错误的消息到更加直接地集成工作流概念（如处理被拒绝的信用卡授权），死信交换器功能非常强大，但由于其处在队列自定义参数的次要位置而往往被忽略。

5.4 控制队列

消费者应用程序有许多不同的应用场景。对于某些应用程序，可以接受多个消费者监听同一个队列，而对于其他一些消费者，一个队列应该只有一个消费者。聊天应用程序可以为每个房间或每个用户创建一个队列，这种场景下的队列被认为是临时的，而信用卡处理应用程序可以创建一个始终存在的持久队列。有了这样一些广泛的应用场景，在处理队列时就很难提供可能需要的每个选项。令人惊讶的是，RabbitMQ 在创建队列时几乎为任何应用场景提供了足够的灵活性。

定义队列时，有多个设置可以确定队列的行为。队列至少可以做到以下几点：

- 自动删除自己。
- 只允许一个消费者进行消费。
- 自动过期的消息。
- 保持有限数量的消息。
- 将旧消息推出堆栈。

按照 AMQP 规范，队列的设置是不可变的，意识到这点很重要。一旦你声明了一个队列，你就不能改变用来创建它的任何设置。要更改队列设置，你必须删除队列并重新创建它。

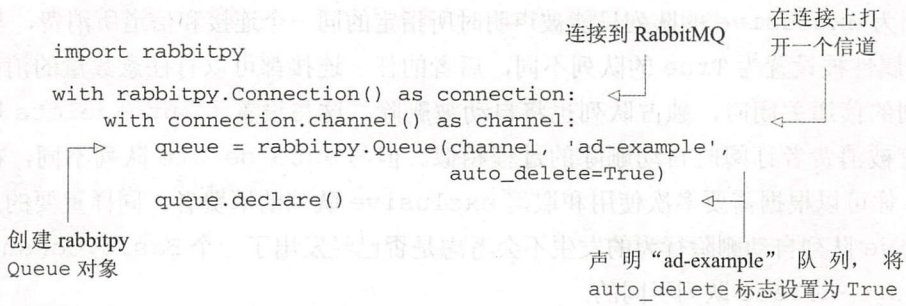
为了探索可用于创建队列的各种设置，我们首先讨论临时队列的相关选项，从队列自己删除自己开始说起。

5.4.1 临时队列

自动删除队列

就像电影碟中谍（Mission Impossible）中的公文包一样，RabbitMQ 提供了一些队列，这些队列一旦使用完就会删除自己并且不会再使用。就像间谍电影中的情报传递工具（dead drop）一样，自动删除的队列可以被创建并且用来处理消息。一旦消费者完成连接和检索消息，在断开连接时队列将被删除。

创建自动删除队列非常简单，只需要在 `Queue.Declare` RPC 请求中将 `auto_delete` 标志设置为 `True` 即可，“5.4.1 自动删除队列” IPython 笔记文件中的例子展示了这一点。



需要注意的是，任意数量的消费者都可以对自动删除队列进行消费；队列只会在没有消费者监听的时候自行删除。可以把队列自动删除当作一种间谍技术，这是一种有趣的应用场景，但这不是自动删除队列的唯一用法。

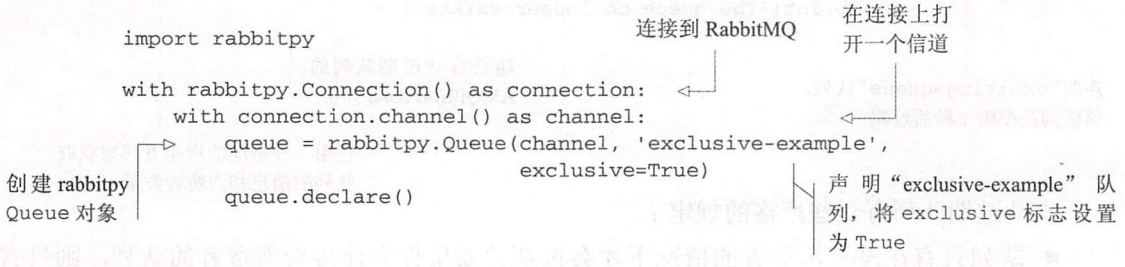
一个应用场景是在线聊天风格的应用程序，其中每个队列代表一个用户的入站聊天缓冲区。如果一个用户的连接断开了，那么这样的应用程序期望该队列和任何未读消息应该被删除，这点是合理的。

另一个示例是 RPC 风格的应用程序。对于向消费者发送 RPC 请求并希望由 RabbitMQ 投递响应的应用程序而言，创建一个在应用程序终止或断开连接时自行删除的队列允许 RabbitMQ 进行自动清理。在这个应用场景中，RPC 回复队列只能由发布原始 RPC 请求的应用程序进行消费，这一点很重要。

只允许单个消费者

如果没有在队列上启用 `exclusive` 设置，RabbitMQ 允许非常随意的消费者行为。它对可以连接到队列并消费消息的消费者数量没有限制。实际上，它鼓励多个消费者，并对能够从队列中接收消息的所有消费者实施轮询（round-robin）投递行为。

在某些情况下，例如 RPC 工作流中的 RPC 回复队列，你需要确保只有单个消费者能够消费队列中的消息。启用队列的独占属性需要在队列创建时传递参数，与 `auto_delete` 参数一样，启用 `exclusive` 属性的队列会在消费者断开连接后自动删除队列。“5.4.1 独占队列”笔记文件的以下示例演示了这一点。



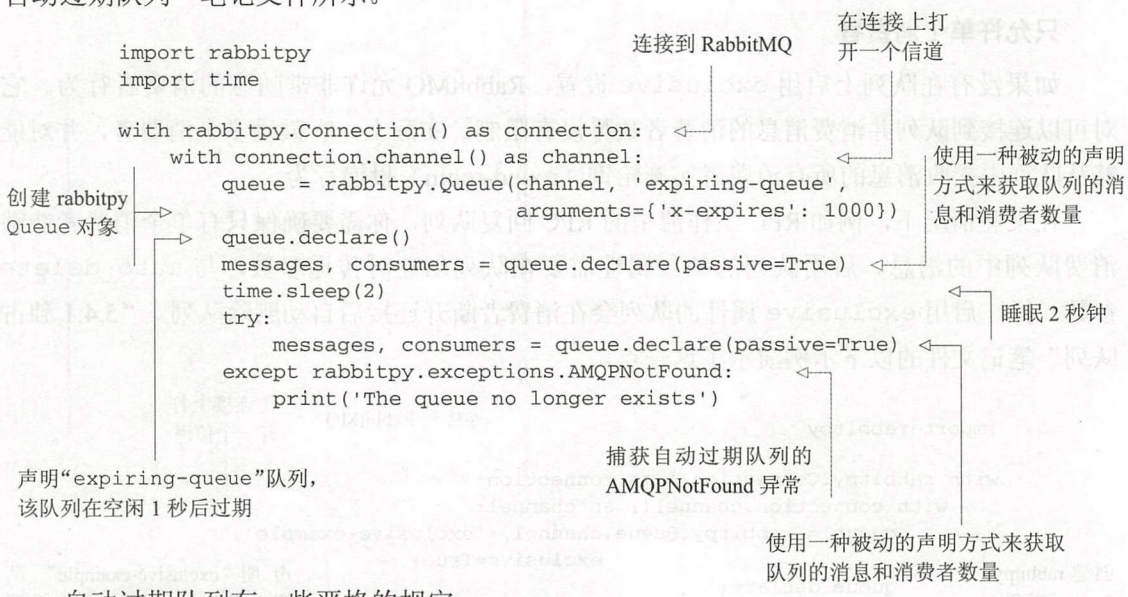
声明为 exclusive 的队列只能被声明时所指定的同一个连接和信道所消费，与 auto_delete 属性被设置为 True 的队列不同，后者的任一连接都可以有任意数量的消费者。当创建队列的信道关闭时，独占队列也将自动被删除，这与设置了 auto-delete 属性的队列在没有被消费者订阅时自动删除的过程相似。但与 auto_delete 队列不同，在信道关闭之前，你可以根据需要多次使用和取消 exclusive 队列的消费者。同样重要的是要注意 exclusive 队列自动删除行为的发生不会考虑是否已经发出了一个 Basic.Consume 请求，这是与 auto-delete 队列不同的。

自动过期队列

当我们讨论自动删除队列时，RabbitMQ 允许在声明一个队列时使用一个可选的参数，这个参数会告诉 RabbitMQ，如果一段时间内没有使用该队列就删除它。就像自动删除的独占队列一样，很容易想象可以将自动过期的队列用作 RPC 回复队列。

假设你有一个对时间敏感的操作，而且你不想无限期地等待 RPC 回复。你就可以创建一个具有 expiration 值的 RPC 回复队列，当该队列过期时就会被删除。使用一种被动的队列声明方式，你就可以轮询队列的存在，并在你看到有消息挂起或队列不再存在时采取动作。

创建一个自动过期的队列非常简单，要做的事情就是使用 x-expires 参数声明一个队列，该参数以毫秒为单位设置队列的生存时间（Time To Live，TTL），如本例中的“5.4.1 自动过期队列”笔记文件所示。



自动过期队列有一些严格的规定：

- 队列只有在没有消费者的情况下才会过期。如果你有连接着消费者的队列，则只有

在发出 Basic.Cancel 请求或断开连接之后才会自动将其删除。

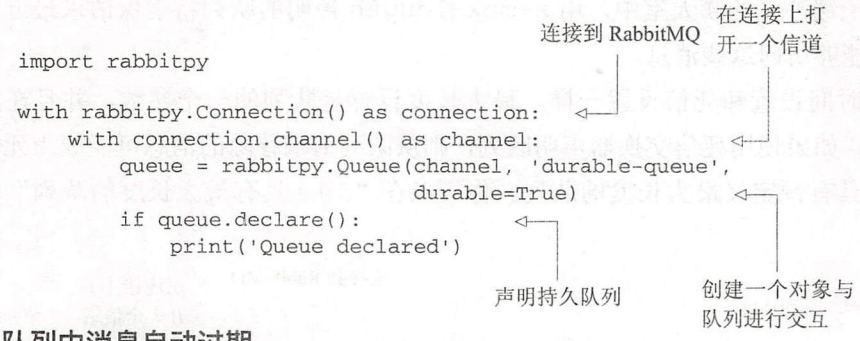
- 队列只有在 TTL 周期之内没有收到 Basic.Get 请求时才会到期。一旦一个 Basic.Get 请求中已经包含了一个具有过期值的队列，那么过期设置无效，该队列将不会被自动删除。
- 与任何其他队列一样，不能重新声明或更改 x-expires 的设置和参数。如果能够重新声明队列，然后用 x-expires 参数的值延长过期时间，那么你将违反 AMQP 规范中的硬性规则，即客户端不得尝试用不同的设置重新声明队列。
- RabbitMQ 不保证删除过期队列这一过程的时效性。

5.4.2　永久队列

队列持久性

当声明那些在服务器重新启动之后仍然存在的队列时，应将 durable 标志设置为 True。队列的持久性经常会与消息的持久化相混淆。正如我们在第 4 章中所讨论的那样，当消息发布时将 delivery-mode 属性设置为 2 时，消息就会存储在磁盘上。相反，durable 标志告诉 RabbitMQ 希望队列被配置在服务器中，直到 Queue.Delete 请求被调用为止。

尽管 RPC 风格的应用程序通常需要队列配合消费者进行创建和删除，但持久队列对于有些应用程序工作流而言非常方便，在这些工作流中多个消费者连接到相同队列，而且路由和消息流不会动态变化。“5.4.2 持久队列”笔记文件演示了如何声明一个持久队列。

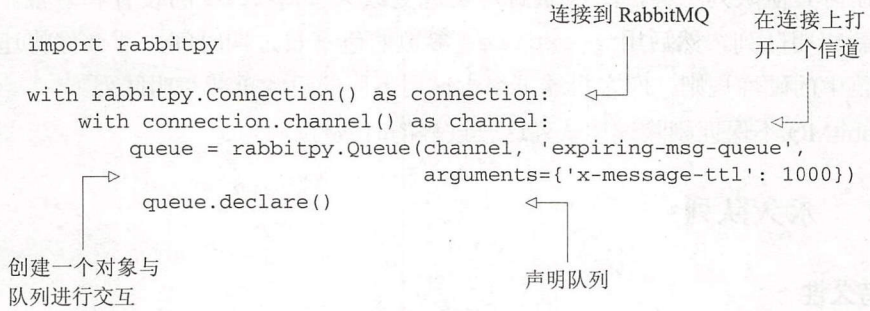


队列中消息自动过期

对于不是很重要的消息，如果它们在没有被消费的状态下存在太久，有时最好让它们自动消失。无论你是对过期后应该删除的过时数据进行分析处理，或者确保可以轻松恢复那些因为高访问量队列而发生宕机的消费者应用程序，消息级别的 TTL 设置允许服务器端对消息的最大生存时间进行限制。声明队列时同时指定死信交换器和 TTL 值将导致该队列中

已到期的消息成为死信消息。

与消息的过期时间属性（可能因消息而异）相反，x-message-ttl 队列设置强制规定了队列中所有消息的最大生存时间。“5.4.2 带消息 TTL 的队列”笔记文件中的以下示例演示了这一点。

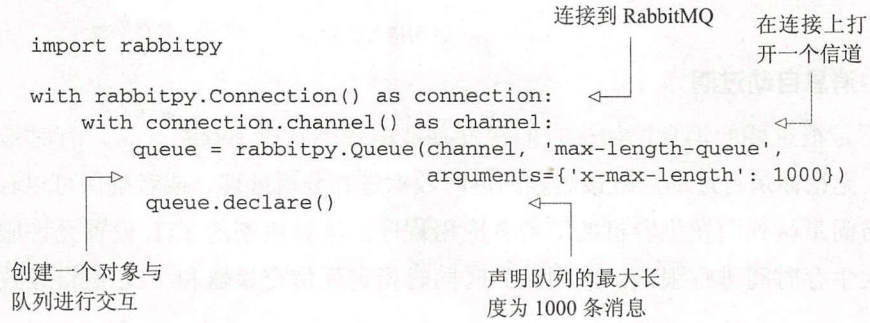


在队列中使用消息级别的 TTL 为消息提供了固有的价值，而这些消息可能对不同的消费者具有不同的价值。对于一些消费者来说，消息可能持有与金钱相关的事务价值，并且必须应用于客户账户。创建一个自动使消息过期的队列将阻止实时仪表盘监听队列从而避免接收到过时信息。

最大长度队列

从 RabbitMQ 3.1.0 开始，可以在声明队列时指定最大长度。如果在队列上设置了 x-max-length 参数，一旦达到最大值，RabbitMQ 会在添加新消息时删除位于队列前端的消息。在具有回滚缓冲区的聊天室中，用 x-max-length 声明的队列将确保请求最近 *n* 个消息的客户端总能够访问这些消息。

像消息过期时间设置和死信设置一样，最大长度设置是队列的一个参数，并且在声明之后不能被改变。如果使用死信交换器声明队列，则从队列前端移除的消息可能成为死信。以下示例显示了具有预定义最大长度的队列。你可以在“5.4.2 具有最大长度的队列”笔记文件中找到它。



5.4.3　任意队列设置

针对队列，RabbitMQ 团队提供了扩展 AMQP 规范的新功能，使用队列参数来传递每个功能集的设置。队列参数可用于设置高可用性队列、死信交换器，消息过期时间、队列过期时间和队列最大长度。

AMQP 规范将队列参数定义为一个表，其中参数值的语法和语义将由服务器确定。RabbitMQ 保留了这些参数（列在表 5.1 中），并忽略了传入的其他任何参数。参数可以是任何有效的 AMQP 数据类型，可以用于任何你喜欢的目的。就我个人而言，我发现参数是设置队列级别监控和阈值的有用方法。

表 5.1　队列的保留参数

参 数 名	目　的
x-dead-letter-exchange	一种交换器，用于路由那些不重新发送且被拒绝的消息
x-dead-letter-routing-key	用于死信消息的可选路由键
x-expires	队列在指定的毫秒数后被删除
x-ha-policy	创建 HA 队列时，指定跨节点实现 HA 的模式
x-ha-nodes	HA 队列分布的节点（见 4.1.6 节）
x-max-length	队列的最大消息数
x-message-ttl	以毫秒为单位的消息过期时间，队列级别执行
x-max-priority	启用最大优先级值为 255（RabbitMQ3.5.0 及更高版本）的队列优先排序功能

5.5　小结

对你的 RabbitMQ 消费者应用程序进行性能优化需要执行基准测试并考虑高速吞吐量和可靠消息投递之间的权衡（就像优化消息发布一样）。在着手编写消费者应用程序时，请考虑以下问题以便为你的应用程序找到最佳选择：

- 你需要确保收到所有的消息，还是可以丢弃消息？
- 你能以批处理操作的方式接收消息并确认或拒绝它们吗？
- 如果没有，你是否可以使用自动批量处理独立操作的事务方式来提升性能？
- 你真的需要消费者的事务提交和回滚功能吗？

- 你的消费者是否需要独占访问消费队列中的消息？
- 消费者遇到错误时应该发生什么？消息是否应该丢弃？重发？或者成为死信消息？

这些问题提供了创建一个可靠消息架构的起点，可以帮助你强化消息发布者和消费者应用程序之间的契约。

现在你已经掌握了发布消息和消费消息的基础知识，我们将在第 6 章探讨如何使用几种不同的消息通信模式和应用场景来将它们付诸于实践。

第 6 章　消息路由模式

本章概要：

- RabbitMQ 中四种基本类型交换器以及交换器插件
- 如何为应用架构选择合适的交换器
- 交换器间路由为消息通信带来更多路由选择

也许 RabbitMQ 最强大之处就在于其灵活性：它能根据消息发布方提供的路由信息，将消息路由到不同的队列中去。不论是将消息发往单个队列、多个队列、交换器，还是另一个由交换器插件提供的外部源，RabbitMQ 的路由引擎能始终保持极快的速度和高度灵活性。即使在一开始构建应用时并不需要这些复杂的路由逻辑，正确选择交换器类型仍然会对应用架构造成举足轻重的影响。

本章将介绍交换器的四种基本类型，以及架构类型：

- Direct 交换器。
- Fanout 交换器。

- Topic 交换器。
- Headers 交换器。

我们将从 direct 交换器 (direct exchange) 开始讲起。之后，我们将使用 fanout 交换器 (fanout exchange) 发送图像给面部识别消费者和图像哈希消费者。我们将使用 topic 交换器 (topic exchange) 基于路由键中的通配符匹配来有选择地路由消息。headers 交换器 (headers exchange) 则是另一种使用消息本身进行消息路由的方式。我将破除交换器的性能神话，向你展示交换器间路由绑定是如何演绎《盗梦空间》般消息路由的。最后，我们将介绍一致性哈希交换器 (consistent-hashing exchange)。当你需要消费者吞吐量的增长超越了单队列、多消费者所提供的能力时，这种交换器类型插件就能一展拳脚。

6.1 通过 direct 交换器路由消息

当需要投递的消息有一个确定的目标（或者多个目标）时，direct 交换器就能派上用场。任何绑定在交换器上的队列，只要它的路由键和发布消息时的一致，它就能收到消息。对于 direct 交换器来说，RabbitMQ 在检查绑定时会比较字符串是否相等。此时不允许使用任意类型的模式匹配（见图 6.1）。

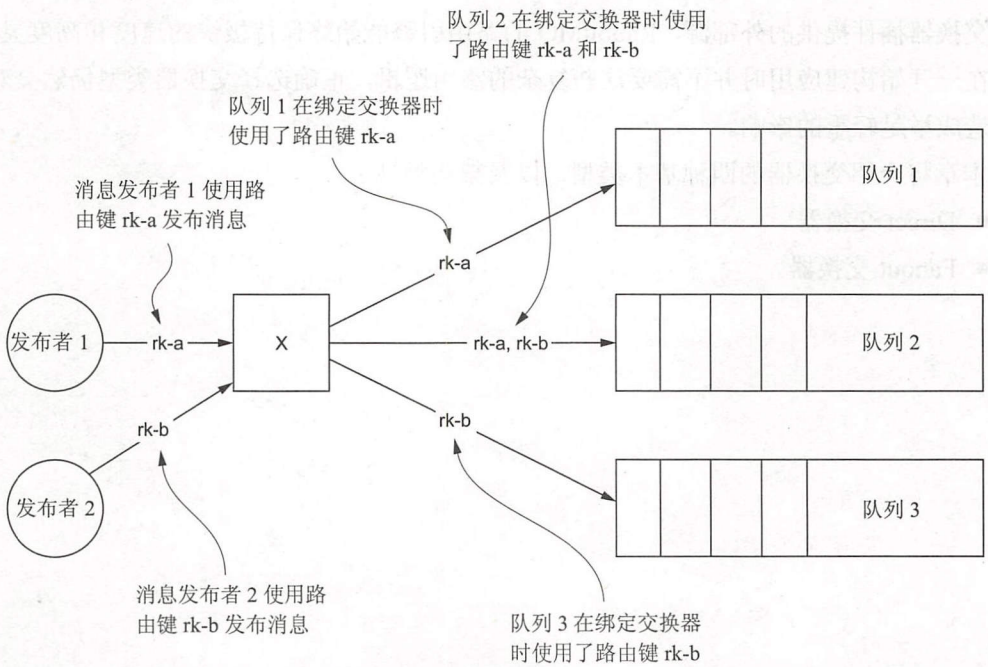
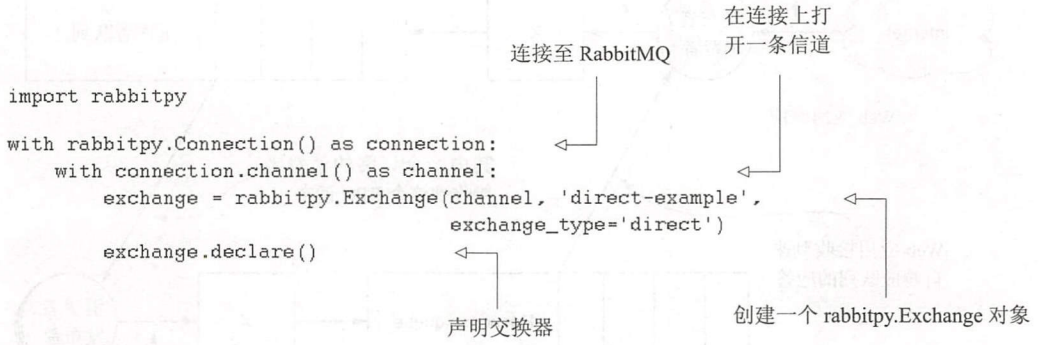


图 6.1 图中使用了 direct 交换器，消息发布者 1 发布的消息将会路由至队列 1 和队列 2，而消息发布者 2 发布的消息将会路由至队列 2 和队列 3 中

就像在图 6.1 中展示的那样，多个队列能够使用相同的路由键绑定到一个 direct 交换器上。每个使用相同路由键绑定的队列能够收到所有使用该路由键发布过来的消息。

RabbitMQ 内置了 direct 交换器类型，所以不需要额外的插件。要创建一个 direct 交换器非常简单，只需将其声明为 direct 即可。可以在“6.1 Direct Exchange”笔记文件中找到这部分代码片段展示。



正是 direct 交换器这种简单性，它非常适用于 RPC 消息通信模式下的路由应答消息。对于那些需要使用由多台服务器提供的不同组件的应用来说，使用 RPC 来解除应用耦合正是实现高度可扩展性的良方。

这是我们基础架构的第一个例子。接下来，你将要编写一个 RPC 工作者，用来消费图片来进行面部识别，然后将结果发回给消息发布方用去。对于那些涉及图像、视频等复杂计算处理过程的应用程序来说，采用远程 RPC 工作者这一模式将使得应用更具可扩展性。假设应用程序运行在云端，请求发布方可以部署在小型虚拟机上，而图像处理工作者则可以利用更高性能的硬件。如果工作负荷支持的话，可以交由 GPU 来处理。

为要开始构建示例应用程序，你将要编写工作者，它是专门用来处理图片的消费者。

6.1.1　创建应用架构

假设你想要实现一个基于 Web 的 API 服务，用于处理由移动电话上传的图片。图 6.1 所描绘的模式可以采用 Tornado Web 框架 (http://tornadoweb.org) 或者 Node.js (http://nodejs.org) 来实现一个轻量级、高度可扩展性、异步的前端 Web 应用。当这个前端应用程序启动的时候，它将在 RabbitMQ 中创建一个唯一名字的队列，用于处理 RPC 响应。

正如图 6.2 所描述的那样，当移动客户端应用程序上传图像时，应用程序就会接收到消息内容并开始处理请求。之后，应用程序会用唯一 ID 来标识远程请求并创建一条消息。当

图像被发布到交换器上时，响应队列的名称将被设置为消息属性的 reply-to 字段，同时请求的 ID 将被放置在 correlation-id 字段中。消息体仅包含不透明的二进制图片信息。

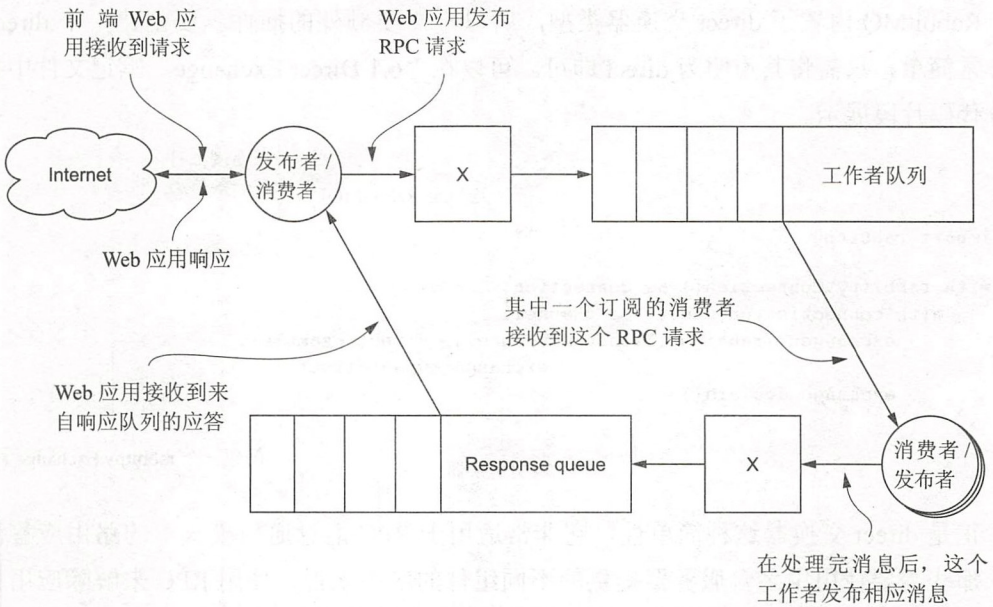


图 6.2　上图描绘了一个简单的 RPC 模式。消息发布方使用 direct 交换器发送请求，同时工作者消费该消息，并将结果发还给最初的发布者

我们在第 2 章已经讨论过底层的帧结构。让我们来回顾下创建 RPC 请求时的帧结构是怎样的（见图 6.3）。

在图 6.3 中，reply-to 和 correlation-id 字段值存放在 Content-Headers 属性载荷中。发送过来的消息体中的图片被划分为三块，以 AMQP 的 body 帧发送。RabbitMQ 最大帧大小为 131,072 字节。这意味着任何消息体只要超过了这个数字，就必须在 AMQP 协议级别分块。由于预先分配的 7 字节必须考虑在内，因此每个消息体帧只能承载 131,065 字节的不透明图片数据。

在第 5 章中曾讲到，当消息发布后并路由到队列上，那么队列上的消费者就能够消费这条消息。消费者可以运行繁重的处理任务，可以执行阻塞的、耗费大量计算能力的或者是 IO 密集型的操作。如果让前端 Web 应用程序来处理的话，势必会阻塞其他客户端。通过将这些计算密集型或者 IO 密集型的任务转交给消费者，前端应用就拥有了异步处理的能力，它能自由地处理其他客户的请求，同时等待来自 RPC 工作者的应答。一旦工作者完成了图像处理，就会将结果以 RPC 请求的形式发回给 Web 前端。这使得 Web 前端能够将处理结

图 6.3　从底层帧结果来分析包含 385,911 字节图像数据的 RPC 消息

果回应远程移动端客户，以完成客户最初发起的请求。

我们不会实现下列清单描述的完整 Web 应用，而仅编写一个简单的消费者处理程序。它尝试对发来的图像进行脸部检测识别，然后在图像上检测到的面部周围标注线框并发回给消息发布者。为了演示发布者是如何操作的，我们将要编写的消息发布者，会把预先指定的目录下每张图片都派发一个处理请求。从图 6.4 中看到，将要实现的简化工作流程包含了应用程序的大部分，仅仅缺少了 Web 应用程序那块内容。

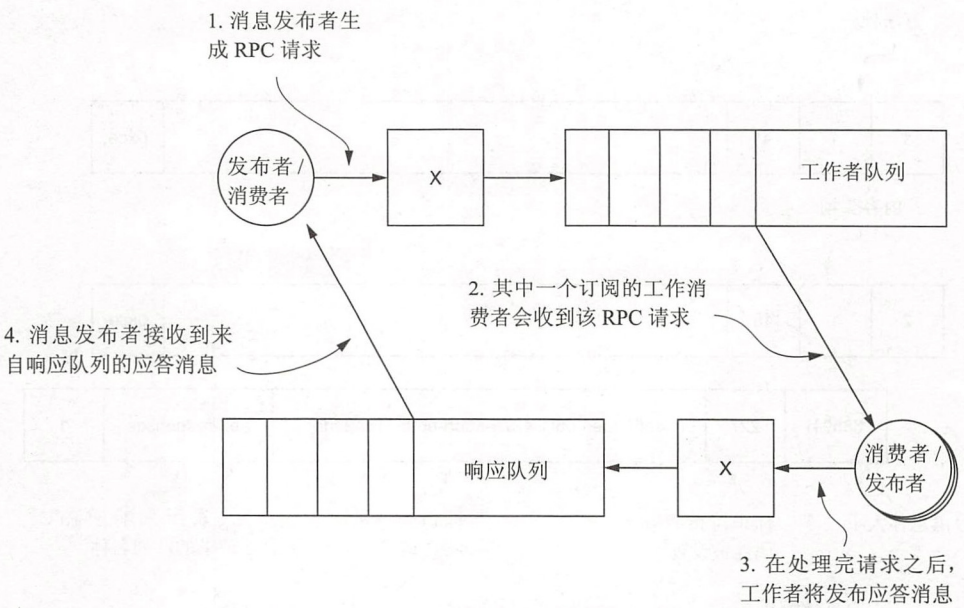
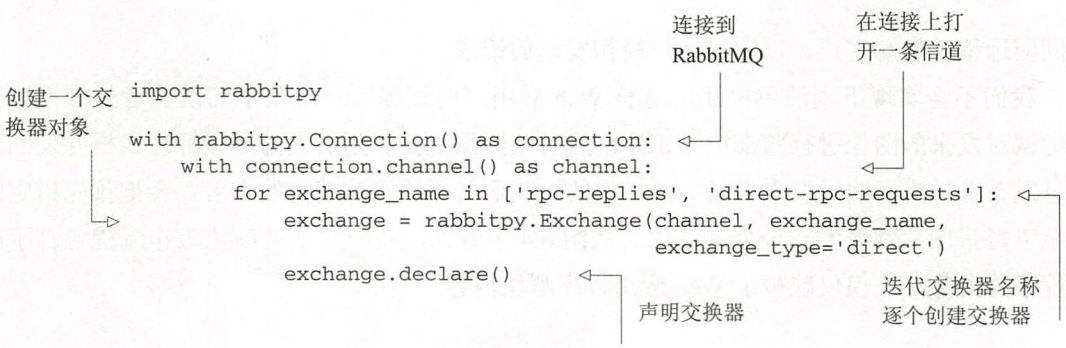


图 6.4　本节内容中实现的简化应用流程

在勾勒出应用结构之后，我们在着手编码前还需要做些准备工作。

声明交换器

在编写消息消费者和消息发布者之前，需要声明一些交换器。在下列代码中，我们没有指定 URL，因此应用程序将使用默认 URL amqp://guest:guest@localhost:5672/%2F 连接到 RabbitMQ。一旦连接上之后，程序会为 RPC 请求的发送和应答分别声明一个交换器。下列代码中声明了 direct 类型 RPC 交换器，它包含在“6.1 RPC Exchange Declaration”笔记文件中。



不同于之前声明交换器的示例，这份代码声明了多个交换器。为了限制实现该任务的

代码量，我们在这里使用了 Python 的 `list` 数组类型来存放交换器的名称。在循环迭代过程中，创建了 rabbitpy `Exchange` 对象，并声明进行了声明。

一旦为我们的 RPC 工作流声明好 RPC 交换器之后，接下来的工作是创建 RPC 工作者。

6.1.2　创建 RPC 工作者

我们将编写 RPC 工作者。作为消费者应用程序，它将接受包含图像文件的消息，并对其进行图像识别处理。我们将采用优秀的 OpenCV (http://opencv.org)，在照片中的每张面部周围画上线框。一旦图像处理完成，新的图像将采用原始消息中的路由信息，被发回到 RabbitMQ 中。

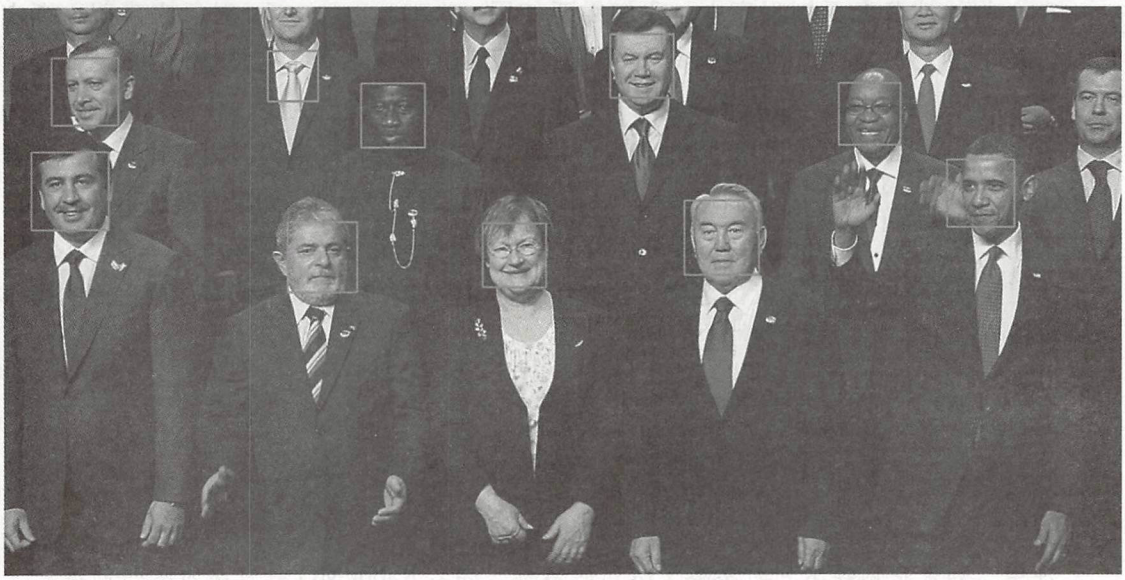


图 6.5　这是一张经过 RPC 面部识别工作者处理过的照片

该 RPC 消息消费者比以往我们见过的示例程序都要复杂，我们多花点时间是值得的。为了不受面部识别细节的牵绊，所有实现面部识别的代码都导入到 Python 包 `ch6` 中的名为 `detect` 的模块中。此外，`ch6.utils` 模块提供了相应的功能来管理磁盘上的图片文件，以便消费者程序的使用。消费者代码包含在“6.1.2 RPC Worker”笔记文件中。

导入合适的库

为了开始构建面部识别消费者程序，首先必须导入应用程序所需的 Python 包或者模块。这些模块包含之前提到的 `ch6` 包、`rabbitpy`、`os` 和 `time`。

```
import os
import rabbitpy
import time  from ch6 import
detect from ch6 import utils
```

os 包用来从磁盘上移除图像文件以及获取当前进程的 ID。time 包则提供了时间信息，用来在系统控制台上输出处理信息。

连接、声明并绑定队列

在引入适当的包之后，就能使用 rabbitpy 来连接 RabbitMQ 并打开信道了：

```
connection = rabbitpy.Connection()
channel = connection.channel()
```

在之前的消费者示例程序中，我们需要使用 rabbitpy.Queue 对象来声明、绑定 RabbitMQ 队列，并从中消费消息。不同于之前的示例，现在这个队列是临时性的、专门用于消费者应用程序的单一实例。我们将 auto_delete 标识设置为 True，并将 durable 标识设置为 False，以便让队列在消费者应用程序结束时会自动消失。为了让其他消费者无法访问该队列上的消息，我们将 exclusive 标识设置为 True。如果有其他消费者尝试从该队列消费消息的话，RabbitMQ 会进行阻止并向其发送一个 AMQP Channel.Close 帧。

为了给队列起一个有意义的名字，我们在字符串中包含 Python 消费者应用程序的操作系统进程 ID，这样就一目了然了。

```
queue_name = 'rpc-worker-%s' % os.getpid()
queue = rabbitpy.Queue(channel, queue_name,
                       auto_delete=True,
                       durable=False,
                       exclusive=True)
```

注意 如果在创建队列时忽略了队列的名称，那么 RabbitMQ 会自动为队列创建名字。你应当能在 RabbitMQ 的管理界面中看到这些队列，它们的名字遵循类似 amq.gen-oCv2kwJ2H0KYxIunVI-xpQ 的模式。

在创建完 Queue 对象后，我们向 RabbitMQ 发送 AMQP 的 Queue.Declare RPC 请求。AMQP 的 Queue.Bind RPC 请求会将队列绑定到合适的交换器。我们使用的是 detect-faces 路由键，这样就能只获取发来的面部识别 RPC 请求了。消息发布者程序会在下一节中介绍。

```
if queue.declare():
    print('Worker queue declared')
if queue.bind('direct-rpc-requests', 'detect-faces'):
    print('Worker queue bound')
```

消费 RPC 请求

在创建并绑定队列后，应用程序就准备开始消费消息了。消费者将使用 rabbitpy.Queue.consume_messages 迭代器方法来消费消息。该方法同时还扮演了 Python 上下文管理器。Python 上下文管理器是一种语言结构，通过 with 语句来调用。对于提供上下文管理器支持的对象来说，它使用 with 语句定义了魔法方法（__enter__ 和 __exit__）。这两个方法分别会在进入代码块和退出代码块时执行。

使用上下文管理器让 rabbitpy 处理发送 Basic.Consume 和 Basic.Cancel 这两个 AMQP RPC 请求，以便你可以专注于自己代码：

```
for message in queue.consume_messages():
```

在遍历 RabbitMQ 投递的每条消息时，通过查询消息的 timestamp 属性以便展示消息被消费之前在队列中存放的时长。消息发布者的代码会在每条消息上自动设置该值，在 RabbitMQ 之外提供信息来源用来记录消息首次创建和发布的时间。

由于 rabbitpy 会自动将属性 timestamp 转换为 Python 的 datetime 对象，所以消费者需要将值转换回 UNIX 时间戳，用来计算消息发布后经过的秒数：

```
duration = (time.time() -
            int(message.properties['timestamp'].strftime('%s')))
print('Received RPC request published %.2f seconds ago' %
      duration)
```

处理图片消息

接下来为了实现面部识别这一功能，消息体中包含的图片文件必须写入磁盘。由于这些文件仅作临时之用，图片将被写入到临时文件中。同时，传入的 content-type 消息属性将用于计算文件的扩展名。

```
temp_file = utils.write_temp_file(message.body,
                                  message.properties['content_type'])
```

在写入文件系统之后，消费者就能使用 ch6.detect.faces 方法来实现面部识别了。该方法返回磁盘上新建的文件路径，文件包含了原始图片并用方框标注识别出的面部：

```
result_file = detect.faces(temp_file)
```

发回结果

最困难的工作已经完成，现在是时候将 RPC 请求的结果发回给最初的发布者了。首先构造响应消息的属性，它包含了原始消息的 correlation-id，这样做可以让发布者知道响应消息关联的是哪张图片。此外，headers 属性设置了消息首次发布的时间戳，这使得发布者可以测量从请求到响应的总时间，将来可以用于监控。

```
properties = {'app_id': 'Chapter 6 Listing 2 Consumer',
              'content_type': message.properties['content_type'],
              'correlation_id':
                 message.properties['correlation_id'],
              'headers': {
                  'first_publish':
                     message.properties['timestamp']}}
```

在设置了响应属性后，从磁盘上读取图片处理结果文件，同时将原始临时文件和结果文件从文件系统中移除：

```
body = utils.read_image(result_file)
os.unlink(temp_file)
os.unlink(result_file)
```

最后，该创建并发布响应消息了。然后确认最初的 RPC 请求消息，以便 RabbitMQ 将其从队列中移除：

```
response = rabbitpy.Message(channel, body, properties)
response.publish('rpc-replies', message.properties['reply_to'])
message.ack()
```

运行消费者应用程序

消费者已编码完成，现在就可以运行了。就像之前的示例程序那样，可以选择 Cell > Run All 而不必单独运行每一个程序。注意，IPython 笔记文件中应用程序最后单元会一直运行下去，直到主动停止它为止。显示 Kernel Busy 指示符说明程序正在运行中（见图 6.6）。你可以保持该浏览器标签页处于打开状态，然后切回到 IPython 仪表板。

6.1.3 编写简单的 RPC 发布者

处于运行之中的 RPC 消费者应用程序会接收消息，并进行面部识别处理，然后返回结果。现在是时候来编写应用程序来发布消息了。在这一场景中，我们的目标是将阻塞的、缓慢的

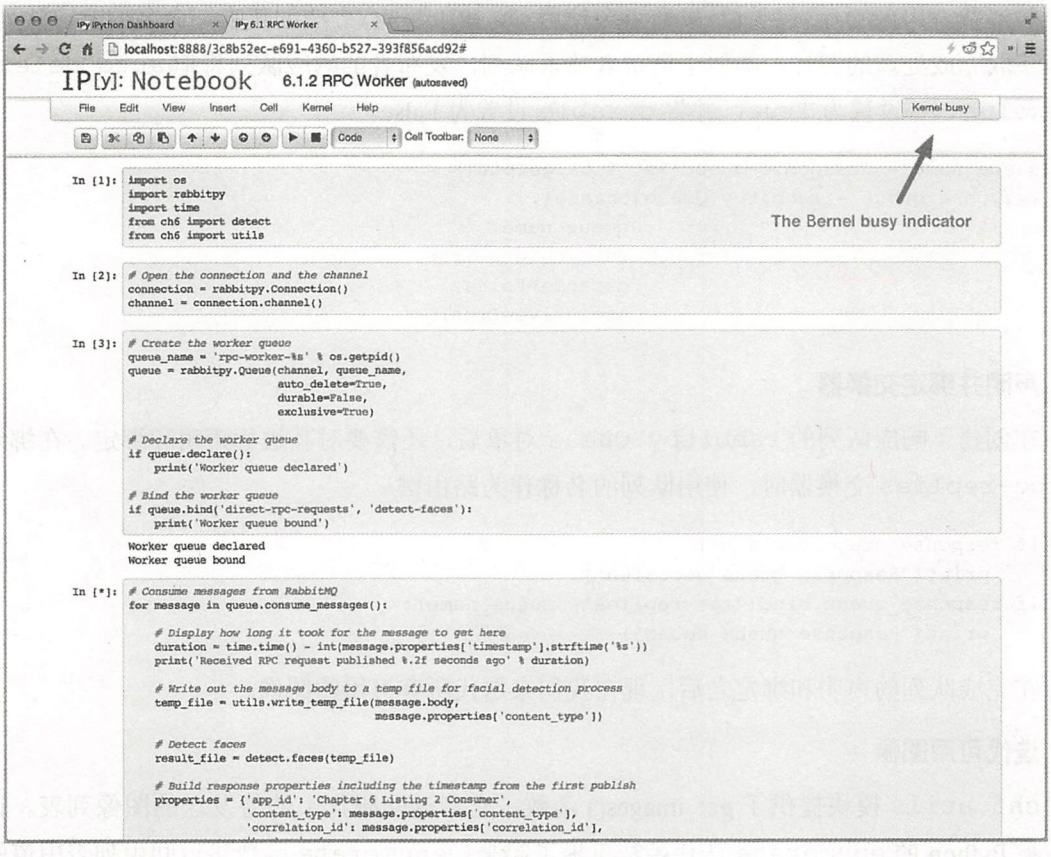


图 6.6　在 IPython notebook 中运行 RPC 工作者

处理过程移至外部消费者，以便 Web 应用程序能够保持高性能，并异步地接收、处理请求，而不用阻塞其他请求，同时对消息的处理也能同时进行。

由于编写完整的异步 Web 应用程序已经超越了本书的范围，此处的发布者代码会简单地发布 RPC 请求消息，并展示接收到的 RPC 响应消息。该示例中的所有图片已经存放于 Vagrant 虚拟机中，并置于公共域中。

确定导入的库

在开始之前，首先必须导入所需的 Python 包和模块，然后才能着手编码。在这个发布者的示例中，除了 `ch6.detect` 模块之外，几乎用到了和之前一样的包和模块。

```
import os
import rabbitpy
import time
from ch6 import utils
```

与消费者用到的 os 包类似，发布者会使用 os.getpid() 方法创建唯一命名的响应队列，用于获取完成处理的图片。类似于消费者请求队列，发布者的响应队列将把 auto_delete 和 exclusive 设置为 True，并将 durable 设置为 False。

```
queue_name = 'response-queue-%s' % os.getpid()
response_queue = rabbitpy.Queue(channel,
                                queue_name,
                                auto_delete=True,
                                durable=False,
                                exclusive=True)
```

声明并绑定交换器

在创建了响应队列的 rabbitpy.Queue 对象后，还需要对其进行声明和绑定。在绑定到 rpc-replies 交换器时，使用队列的名称作为路由键：

```
if response_queue.declare():
    print('Response queue declared')
if response_queue.bind('rpc-replies', queue_name):
    print('Response queue bound')
```

在完成队列的声明和绑定之后，现在我们来迭代所有可用的图像。

迭代可用图像

ch6.utils 模块提供了 get_images() 函数，它将返回磁盘上的待发送的图像列表。该函数被 Python 的 enumerate 迭代函数包裹了起来。enumerate 函数将返回由列表中值的当前下标和关联的值所组成的元组。元组是很常见的数据结构。在 Python 中，它是对象的不可变序列。

```
for img_id, filename in enumerate(utils.get_images()):
```

这块代码逻辑将构造消息并将其发送至 RabbitMQ 中。但在发送之前，让我们打印消息详情，以便知晓将要发布出去进行处理的图片的内容：

```
print('Sending request for image #%s: %s' % (img_id, filename))
```

构造请求消息

创建消息非常简单直白。首先构建 rabbitpy.Message 对象，第一个参数传入信道，然后使用 ch6.utils.read_image() 方法从磁盘上读取原始图像数据，并将其用作消息

体参数。

最后，创建消息属性。使用 ch6.utils.mime_time() 方法返回的 mime 类型来设置消息的 content-type。使用 enumerate 迭代函数提供的 img_id 值来设置消息的 correlation-id 属性。在异步的 Web 应用程序中，可以使用客户端的连接 ID 或者套接字文件描述符编号来代替。最后，将消息的 reply_to 属性设置为发布者的响应队列名称。如果将 opinionated 标签设置为 True 的话，那么 rabbitpy 库将会在 timestamp 属性缺失时，自动对其进行设置。

```
message = rabbitpy.Message(channel,
                           utils.read_image(filename),
                           {'content_type':
                                utils.mime_type(filename),
                            'correlation_id': str(img_id),
                            'reply_to': queue_name},
                           opinionated=True)
```

在创建完消息对象后，现在使用 detect-faces 作为路由键将其发送至 direct-rpc-requests 交换器：

```
message.publish('direct-rpc-requests', 'detect-faces')
```

一旦运行起来，消息将被发送出去，并且很快会被 RPC 消费者应用程序接收到。

等待应答

在异步 Web 应用程序中，应用程序会在处理另一位客户请求的同时等待来自 RPC 消费者的响应。为了演示的目的，我们将创建一个同步阻塞的服务器来替代异步服务器。我们的发布者不会以异步的方式在消费响应队列的同时处理其他任务，而是会使用 AMQP RPC 方法 Basic.Get 来检测队列中是否存在消息并进行获取，如下所示：

```
message = None
while not message:
   time.sleep(0.5)
   message = response_queue.get()
```

确认消息

一旦收到消息之后，发布者应当对其进行确认，以便 RabbitMQ 将该消息从队列中移除：

```
message.ack()
```

处理响应

也许你像我一样想要知道面部识别和消息路由花费了多少时间，因此需要添加一行代码来打印从最初的消息发布直到收到响应所经历的总时长。在复杂的应用程序中，我们可以使用消息属性来承载这类元数据信息，用于调试、监控、趋势研究以及数据分析。

```
duration = (time.time() -
            time.mktime(message.properties['headers']['first_publish']))
print('Facial detection RPC call for image %s duration %.2f sec' %
      (message.properties['correlation_id'], duration))
```

代码中打印的持续时间用到了消费者设置在消息属性 header 表中的 first_publish 时间戳。采用这种方式可以计算出从最初的 RPC 请求发布到收到 RPC 响应全程所需的时间。

最后，使用 ch6.utils .display_image() 函数在 IPython 笔记文件中显示图片：

```
utils.display_image(message.body,
                    message.properties['content_type'])
```

清理

我们完成了发布者的代码逻辑，下列代码将关闭信道和连接，不必采用四个空格的缩进：

```
channel.close()
connection.close()
```

测试完整的应用程序

现在开始进行测试。在 IPython 笔记文件中打开“6.1.3 RPC Publisher”，点击 Run Code 按钮，发送消息并观察结果（见图 6.7）。

你也许注意到了，面部识别代码并不完美，但对于性能不强的虚拟机来说，表现还算不错。若想要提升面部识别的效果，可以再额外添加一些算法，以便找出每个算法中都出现的结果。虽然这种实现方式对于实时 Web 应用程序来说太慢了，但是对于那些特殊硬件上的大批 RPC 消费者应用程序来说却非如此不可。幸运的是，RabbitMQ 提供了多种方法来将同一消息路由到不同的队列中去。在下一节中，我们将获取来自 RPC 发布者发来的消息，而不用影响已经建立好的工作流程。要达成这一目的，我们在处理 RPC 请求时将使用 fanout 交换器来替代 direct 交换器。

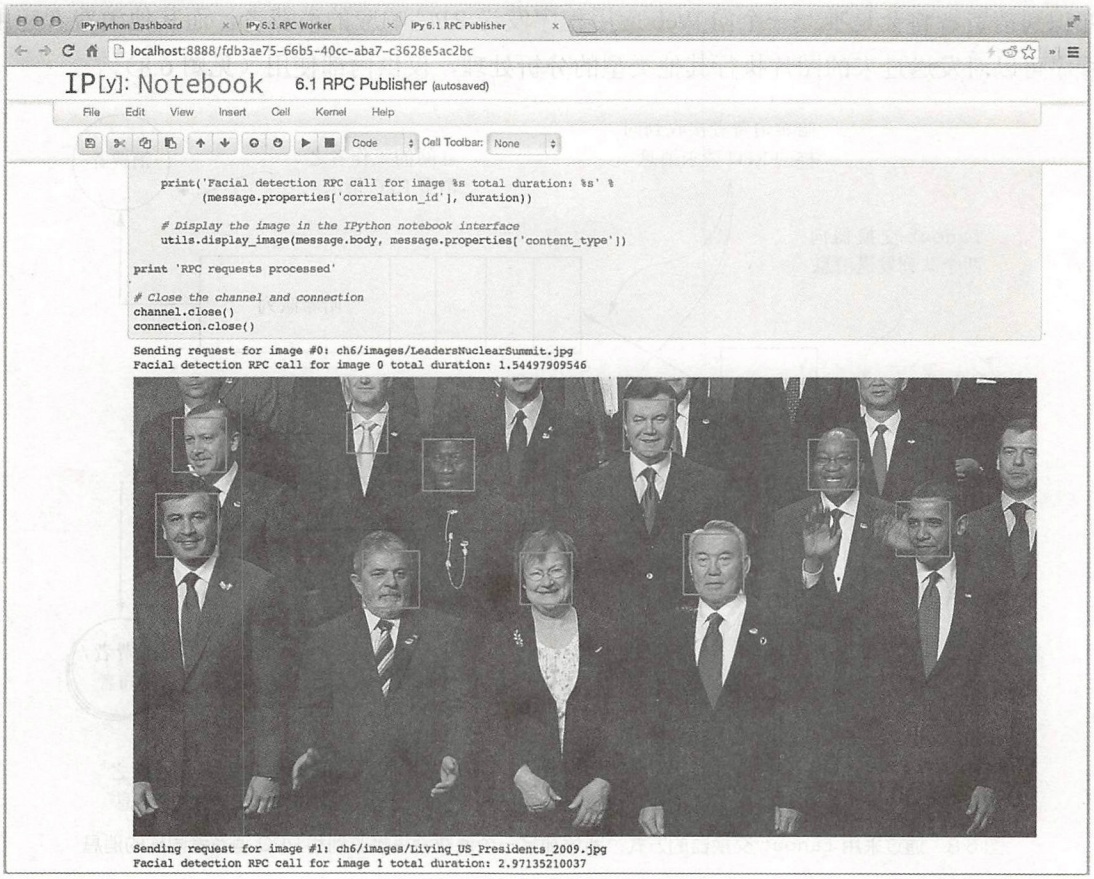


图 6.7　RPC 发布者从消费者接收到的结果

6.2　通过 fanout 交换器广播消息

Direct交换器使得队列能够接收特定目的的消息。不同于此，fanout 交换器并不作区分。所有发往 fanout 交换器的消息会被投递到所有绑定到该交换器上的队列中。由于 RabbitMQ 不需要在投递消息时检测路由键，这将带来可观的性能优势。不过由于缺少选择性，这意味着所有绑定到 fanout 交换器上的队列对应的消费者都应当能够消费发送过来的消息。

假设除了面部检测之外，你还想要为移动应用程序编写工具，用于实时识别垃圾邮件发送者。我们让 Web 应用程序在执行面部识别 RPC 请求时将消息发送至 fanout 交换器，因此你就能将 RPC 消费者队列和其他用于消息处理的消费者应用程序绑定到该交换器上。面

部识别消费者将会是唯一一个向 Web 应用程序发送 RPC 响应的消费者，而其他消费者应用程序可以对发送过来的图片执行其他类型的分析处理，仅供内部使用（见图 6.8）。

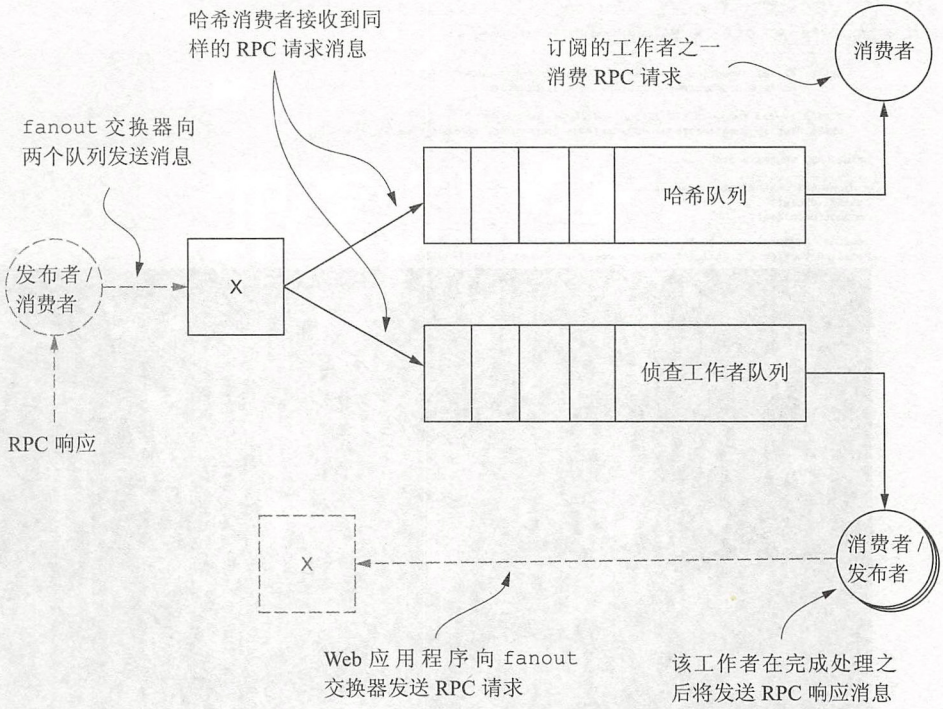


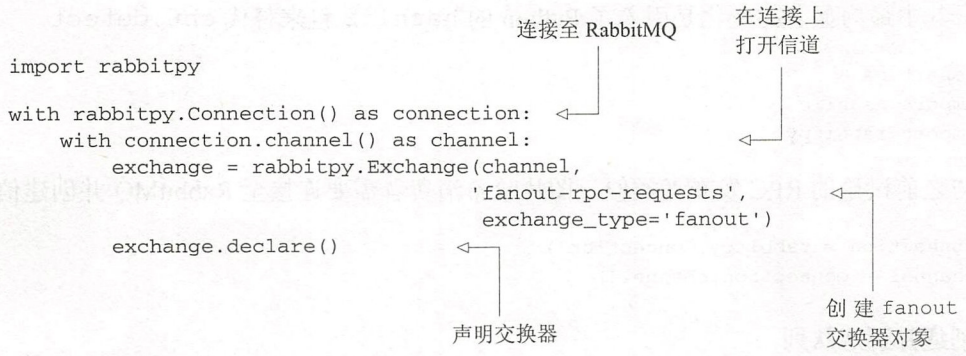
图 6.8　通过采用 fanout 交换器的方式，新添加另一个消费者将接收到和 RPC 消费者同样的消息

根据笔者的经验，垃圾邮件发送者经常会使用同样的图片用于注册服务或者向服务端提交内容。其中一种缓解垃圾邮件攻击的方式是计算图片的指纹，并采用图片指纹数据库来检测垃圾邮件，对拥有相同指纹的新上传的图片采取行动。在下面的代码示例中，我们将采用 RPC 请求消息来触发面部识别消费者来计算图片指纹。

6.2.1　修改面部检测消费者

下面展示的示例程序构建于 6.1 小节中的例子的基础之上，我们做了些细微的改动，添加了图片哈希消费者，它会在 RPC 请求发出时计算图片的哈希值。

首先需要创建 fanout 交换器。下列代码片段来自“6.2.1 Fanout Exchange Declaration”笔记文件。



除此之外，最初来自 6.1 小节的消费者代码需要在队列绑定上进行细微的改动。不同于之前使用路由键绑定到 direct-rpc-request 交换器上，这里消费者需要绑定到 fanout-rpc-requests 上，无需路由键。“6.2.1 RPC Worker”笔记文件已经做了这一改动。它改动了这一行：

```
if queue.bind('direct-rpc-requests', 'detect-faces'):
```

使用了新的 fanout 交换器：

```
if queue.bind('fanout-rpc-requests'):
```

针对发布者代码的变动仅需更改消息发往的交换器即可。同样的，“6.2.1 RPC Publisher”笔记文件已经对此做了修改，它改动了这一行：

```
message.publish('direct-rpc-requests', 'detect-faces')
```

如下，使用了新的交换器：

```
message.publish('fanout-rpc-requests')
```

在发布和消费消息之前，让我们先编写图片哈希，也就是指纹打印消费者吧。

6.2.2　创建一个简单的图片哈希消费者

为了简单起见，消费者将使用简单的二进制哈希算法 MD5。有很多更为复杂的算法能更好地完成图片哈希计算的任务，支持多种剪裁方式、分辨率和位深度功能，但是本示例的重点在于展示 RabbitMQ 交换器而非图片识别算法。

导入基本库并连接至 RabbitMQ

让我们着手开始吧。“6.2.2 Hashing Consumer”笔记文件中的消费者代码和 RPC 消费者类似。其中最为显著的变化是引入了 Python 的 `hashlib` 包来替代 `ch6.detect`。

```
import os
import hashlib
import rabbitpy
```

与之前讨论的 RPC 发布者类似，图片哈希消费者需要连接至 RabbitMQ 并创建信道：

```
connection = rabbitpy.Connection()
channel = connection.channel()
```

创建并绑定队列

在打开信道之后创建队列，该队列会在消费者断开后自动删除，并且对该消费者来说是排他的（`exclusive=True`）。

```
queue_name = 'hashing-worker-%s' % os.getpid()
queue = rabbitpy.Queue(channel, queue_name,
                       auto_delete=True,
                       durable=False,
                       exclusive=True)
if queue.declare():
    print('Worker queue declared')
if queue.bind('fanout-rpc-requests'):
    print('Worker queue bound')
```

图像哈希

消费者代码十分直白。它会迭代每一条收到的消息并创建 hashlib.md5 对象，传入二进制消息数据。之后，它将打印哈希值。输出的那一行可以很容易地被替换为数据库插入或者 RPC 请求。它将哈希值与数据库中已存在的哈希值进行比较。最后，消费者对消息进行确认并等待下一条消息。

```
for message in queue.consume_messages():
    hash_obj = hashlib.md5(message.body)
    print('Image with correlation-id of %s has a hash of %s' %
          (message.properties['correlation_id'],
           hash_obj.hexdigest()))
message.ack()
```

注意 对于存储一套哈希值来说，Redis (http://redis.io) 是极佳的选择。它为哈希查找提供了快速的内存数据结构，并且对此类数据提供了非常快速的查询响应。

测试新的工作流程

在编写完成新的消费者程序并且调整了其他应用程序之后，现在该进行测试了。在浏览器中打开“6.2.2 Hashing Consumer”、“6.2.1 RPC Worker”和“6.2.1 RPC Publisher”笔记文件。首先运行“6.2.2 Hashing Consumer”笔记文件，然后运行“6.2.1 RPC Worker”笔记文件，最后通过运行“6.2.1 RPC Publisher”笔记文件来发送图片以启动整个流程。像前一个章节中运行示例程序那样，你应当看到相同的RPC发布者和消费者应用程序的响应。此外，你应当在“6.2.2 Hashing Consumer”应用程序中看到类似图 6.9 这样的输出。

```
In [*]: # Consume nessages from RabbitMQ
for message in queue.consume_messages():

    # Create the hashing object
    hash_obj = hashlib.md5(message.body)

    # Print out the info, this might go into a database or iog file
    print('Image with correlation-id of %s has a hash of %s' %
          (message.properties['correlation_id'],
           hash_obj.hexdigest()))

    # Acknowlcdgc the delivcry of the RPC request message
    message.ack()

Image with correlation-id of 0 has a hash of ba3acb1d632c42c4a7b7ea0eaf32b50a
Image with correlation-id of 1 has a hash of 3b2bb8455f68943745c3a362a567cba5
```

图 6.9　在 IPython 笔记文件中哈希消费者示例运行结果

Fanout 交换器提供了非常棒的方式，让每一个消费者都能访问到原始数据。不过这也成为了一把双刃剑，因为消费者无法对收到的消息进行选择。假设你想要单一交换器，允许路由不同类型的 RPC 请求，以便执行诸如对每条 RPC 请求进行审计这样的普通任务而不关心类型。在这一场景中，topic 交换器允许 RPC 工作消费者根据自身的任务来绑定路由键。同时对于请求审计消费者来说，通过使用通配符来匹配所有消息或者其中的某个子集。

6.3　使用 topic 交换器有选择地路由消息

类似于 direct 交换器，topic 交换器会将消息路由至匹配路由键的任一队列中。但是通过采用句点分隔的形式，队列可以通过使用基于通配符的模式匹配的方式来绑定到路由键上。通过使用星号（*）和井号（#）字符，你可以在同一时刻匹配路由键的特定部分，甚至是多个部分。星号将会匹配路由键中下一个句点前的所有字符，而井号键将会匹配接下来所有的字符，包括句点。

图 6.10 展示了由三个部分组成的 topic 交换器路由键。第一个部分表明消息应当被路由

图 6.10　topic 交换器路由键由三部分组成

到专门处理图片消息的消费者。第二部分表明该消息包含了新的图片。第三部分包含了额外的数据，用来将消息路由至专门处理资料相关的消费者队列中去。

如果我们想要以图片上传流程为基础，创建基于消息通信的架构来管理网站上所有图片相关的任务的话，那么下面列举的这些路由键可以用来描述其中一部分消息。

- `image.new.profile`——用于表示包含新资料图片的消息。
- `image.new.gallery`——用于表示包含新相册图片的消息。
- `image.delete.profile`——用于表示含有删除资料图片的元数据的消息。
- `image.delete.gallery`——用于表示含有删除相册图片的元数据的消息。
- `image.resize`——用于表示要求调整图片大小的消息。

在之前讲到的示例路由键中，路由键的语义重要性显而易见。它描述了消息的意图或者内容。通过对路由至 topic 交换器的消息的键进行语义化命名，单条消息可以通过路由键的一部分来进行路由，并最终投递到特定的任务队列中去。在图 6.11 中，topic 交换器会根据消费者应用程序队列绑定至交换器的方式来决定哪个队列将会收到消息。

topic 交换器是消息路由的极佳选择，它使得单一用途的消费者能够对消息进行不同的处理。在图 6.11 中，面部检测 RPC 工作者队列绑定到了 `image.new.profile` 上，其表现仿佛绑定到了 direct 交换器上，它将只接受新的资料图片请求。图片哈希消费者绑定到了 `image.new.#` 上，它将接收新图片而不关心其来源。用于维护物化用户目录的消费者则从绑定到 `#.profile` 上的队列消费消息，它将接收所有以 `.profile` 结尾的消息，并执行物化任务。图片删除消息将被发送到绑定在 `image.delete.*` 上的队列，它允许单个消费者删除上传至站点的所有图片。最后，用于审计的消费者绑定在了 `image.#` 上，它将接收到所有图片相关的消息，打印信息以便问题追踪和行为分析。

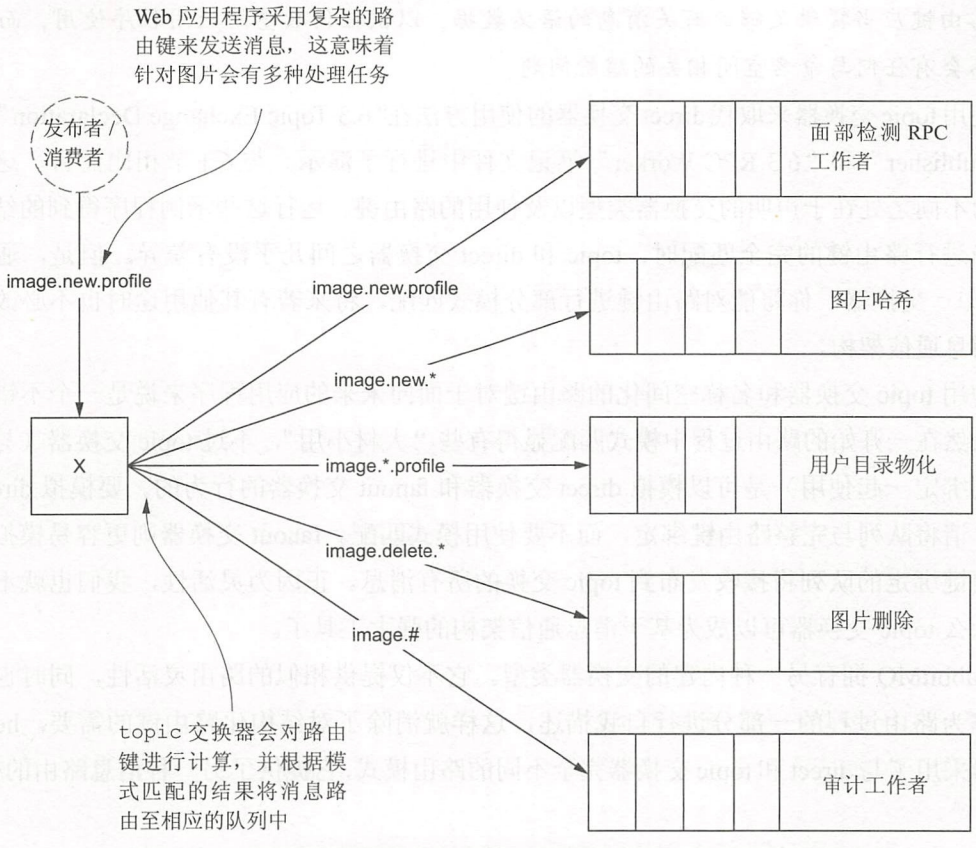


图 6.11　基于路由键的构造方式，消息会被有选择地路由到不同的队列中去

利用这种体系结构的单一用途的消费者可以更容易维护和扩展。这一结论是通过与单一应用程序对投递给单一队列的消息执行相同的操作进行对比得出的。单一应用程序增加了操作和代码复杂性。考虑一下如何使用模块化方法来简化消费者代码以执行复杂操作的，例如移动硬件，通过添加新消费者的方式来提升处理吞吐量，甚至只是添加或删除应用程序功能。通过单一用途的模块化方法以及 topic 交换器，可以由新的消费者和队列来组建合适的新功能，而不会影响其他消费者应用程序的运作。

注意　创建富含消息语义的路由键很有用。它能够描述消息的意图或内容。相对于基于应用程序的细节来设计消息及其路由键来说，基于事件的通用消息通信则鼓励消息的可重用性。当开发人员能够在应用程序中复用现有的消息时，代码复杂性降低以及消息吞吐量的降低是最显而易见的优势。连同虚拟主机和交换器，

路由键应当提供足够的有关消息的语义数据，以供任何数量的应用程序使用，而不会有任何与命名空间相关的尴尬问题。

使用topic交换器来取代direct交换器的使用方法在“6.3 Topic Exchange Declaration”、“6.3 RPC Publisher”和“6.3 RPC Worker”笔记文件中进行了演示。与6.1节相比而言，这当中最大的不同之处在于声明的交换器类型以及使用的路由键。运行这些示例程序得到的结果显示，当进行路由键的完全匹配时，topic和direct交换器之间几乎没有差异。但是，通过使用 `topic` 交换器，你将能对路由键进行部分模式匹配，将来若有其他用途时也不必改动现有的消息通信架构。

使用topic交换器和名称空间化的路由键对于面向未来的应用程序来说是一个不错的选择。虽然在一开始的路由过程中模式匹配显得有些“大材小用”，不过topic交换器（与正确的队列绑定一起使用）是可以模拟direct交换器和fanout交换器的行为的。要模拟direct交换器，请将队列与完整路由键绑定，而不要使用模式匹配。fanout交换器则更容易模拟，因为路由键绑定的队列将接收发布到topic交换的所有消息。正因为灵活性，我们也就不难理解为什么topic交换器可以成为基于消息通信架构的强大工具了。

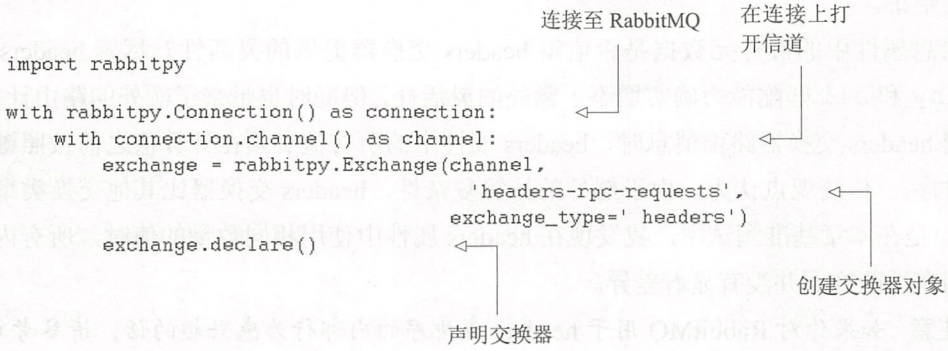
RabbitMQ拥有另一种内建的交换器类型。它不仅提供相似的路由灵活性，同时也允许消息作为路由过程的一部分进行自我描述，这样就消除了对结构化路由键的需要。headers交换器采用了与direct和topic交换器完全不同的路由模式，它提供了另一种消息路由的视角。

6.4 使用 headers 交换器有选择地路由消息

第四种内建交换器类型是headers交换器。它通过采用消息属性中的headers表支持任意的路由策略。绑定至headers交换器的队列会向 `Queue.Bind` 参数中传入键值对数组以及 `x-match` 参数。`x-match` 参数是字符串类型，可以设置为 `any` 或者 `all`。如果将其设置为any，同时 `headers` 表中的值匹配了任何一个绑定值的话，消息就会被路由过去。如果将 `x-match` 设置为 `all` 的话，那么所有传入 `Queue.Bind` 中的参数值必须全部匹配才行。这并不排除消息在 `headers` 表中拥有额外的键值对。

为了演示如何使用headers交换器，我们将修改6.1节中的RPC工作者和发布者示例程序，将路由键移至 `headers` 表值中。不同于使用topic交换器，消息本身包含的值将会用来构造路由条件。

在我们将 RPC 发布者和工作者改造为使用 headers 交换器之前，首先让我们来声明 headers 交换器。下列示例程序将创建一个名为 headers-rpc-requests 的 headers 交换器。这份代码在“6.4 Headers Exchange Declara-tion”笔记文件中。



在声明了交换器之后，让我们来检查一下“6.4 RPC Publisher”笔记文件中的 RPC 发布者代码。主要有两处不同。其一为构造用于发布的消息。在本示例中，消息的 headers 属性将被填充：

```
message = rabbitpy.Message(channel,
                           utils.read_image(filename),
                           {'content_type': utils.mime_type(filename),
                           'correlation_id': str(img_id),
                           'headers': {'source': 'profile',
                                       'object': 'image'
                                       'action': 'new'},
                           'reply_to': queue_name})
```

可以看到，一共为 headers 属性设置了三个值：source、object 和 action。这些值将在消息被发布后用于路由，因而也就不再需要路由键了。因此需要将 message.publish() 修改为消息将要路由前往的 headers 交换器名称。

```
message.publish('headers-rpc-requests')
```

在运行代码之前，让我们检查“6.4 RPC Worker”笔记文件中对 RPC 工作者的更改，并运行代码启动消费者。最主要的更改是 Queue.Bind 的调用。这里没有绑定到路由键上，在调用 Queue.Bind 时指定了匹配类型用于将图片路由至队列，以及用于匹配的所有属性：

```
if queue.bind('headers-rpc-requests',
              arguments={'x-match': 'all',
                         'source': 'profile',
                         'object': 'image',
                         'action': 'new'}):
```

x-match 参数被指定为了 all。这意味着消息 headers 中的 source、object 和 action 值必须和绑定参数指定的值完全匹配。如果你现在运行“6.4 RPC Worker”笔记文件，然后再运行“6.4 RPC Publisher”笔记文件的话，你将能看到和 direct、topic 交换器示例程序一样的结果。

消息属性中的额外元数据是否值得 headers 交换器提供的灵活性？尽管 headers 交换器通过 any 和 all 匹配能力确实增添了额外的灵活性，但同时也带来了额外的路由计算开销。在使用 headers 交换器路由消息时，headers 属性中的所有值必须在计算值之前按照键的名称进行排序。传统观点认为，由于额外的计算复杂性，headers 交换器比其他交换类型要慢得多。但是在本章基准测试中，我发现在 headers 属性中使用相同数量的值时，所有内置交换机之间在性能方面并没有显着差异。

注意 如果你对 RabbitMQ 用于 headers 表排序的内部行为感兴趣的话，请参考 Git 仓库 rabbit-server 中 rabbit_misc 模块的 sort_field_table 函数。GitHub 上的代码地址：https://github.com/rabbitmq/rabbitmq-server/blob/master/ src/rabbit_misc.erl.。

6.5 交换器性能基准

值得注意的是，使用 headers 属性会对消息发布的性能直接造成影响，无关乎消息将要发往的选择器类型（见图 6.12）。

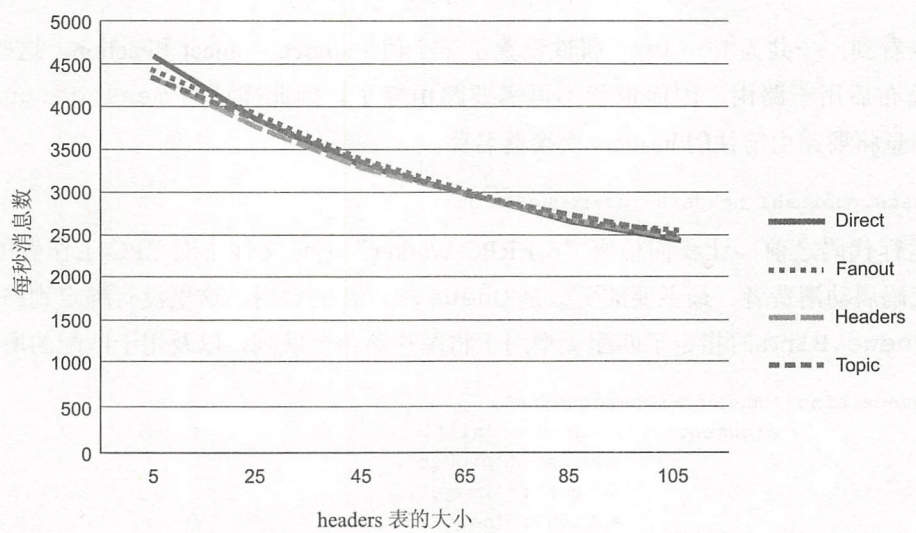


图 6.12 交换器类型和 header 表大小对发布速率影响概览

正如你所看到的那样，四种内建交换器类型的性能比较一致。这一基准显示，在比较拥有相同消息 headers 的消息时，可以发现不管交换器是什么类型，消息发布的速度不会有显著的差异。

让我们再来看看针对 topic 和 headers 交换器的更理想的测试案例吧。在图 6.13 中，通过 topic 交换器发布的消息拥有相同的消息体和空的 headers 表，而通过 headers 交换器发布的消息在 headers 属性中包含了路由键。在这一场景中，当进行两者之间消息路由的基准比较时，显然 topic 交换器要比 headers 交换器拥有更好的性能。

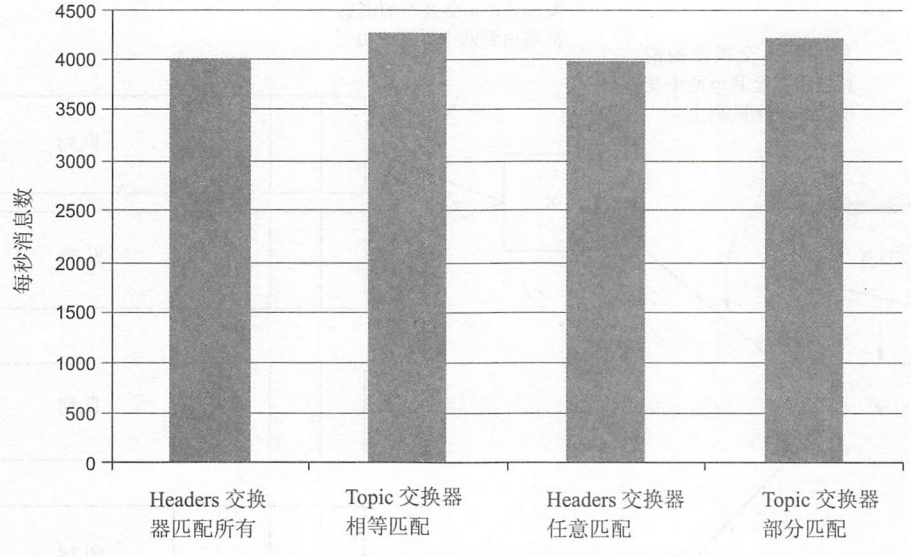


图 6.13　headers 和 topic 交换器的发布速率

如果你使用 headers 属性的话，除非在 headers 属性中使用了非常巨大的表值，否则总体的消息发布速率并不会因选择 headers 交换器而产生巨大影响。这一性能损失对所有内建交换器类型都一样。

在对内建交换器类型的能力以及它们之间的对比有了清晰的认识之后，现在让我们学习如何使用多个类型的交换器来解决发送至 RabbitMQ 的单条消息。

6.6　交换器间路由

如果你认为当前消息路由的灵活性还不够，你的应用程序需要某种交换器类型的一部分功能加上另一种类型的一部分功能来处理同一条消息话，那么 RabbitMQ 也会带给你惊喜。RabbitMQ 团队往 RabbitMQ 中添加了一种非常灵活的机制（未包含在 AMQP 规范中），它

允许你将消息路由至交换器的任意组合。交换器间的绑定机制类似于队列绑定，与将队列绑定至交换器上不同的是，你需要使用 RPC 方法 `Exchange.Bind` 将一个交换器绑定至另一个交换器上。

当使用交换器间绑定时，绑定交换器的路由逻辑和绑定队列是一致的。任何交换器都可以绑定到任何一个内建类型的交换器上。有了这一功能你就可以发挥想象力将交换器连接起来。你是否想过将命名空间化的键路由至 topic 交换器，然后再基于属性 header 表将它们分发出去呢？交换器间绑定正好适用于此场景（见图 6.14）。

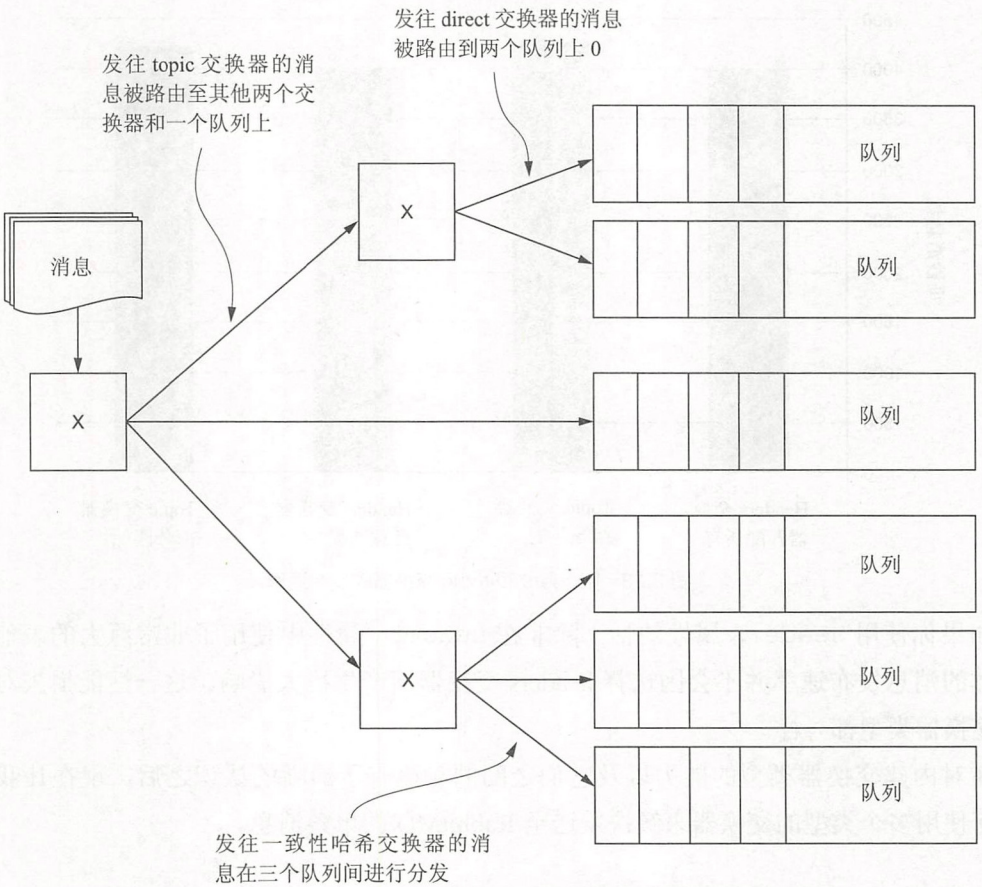
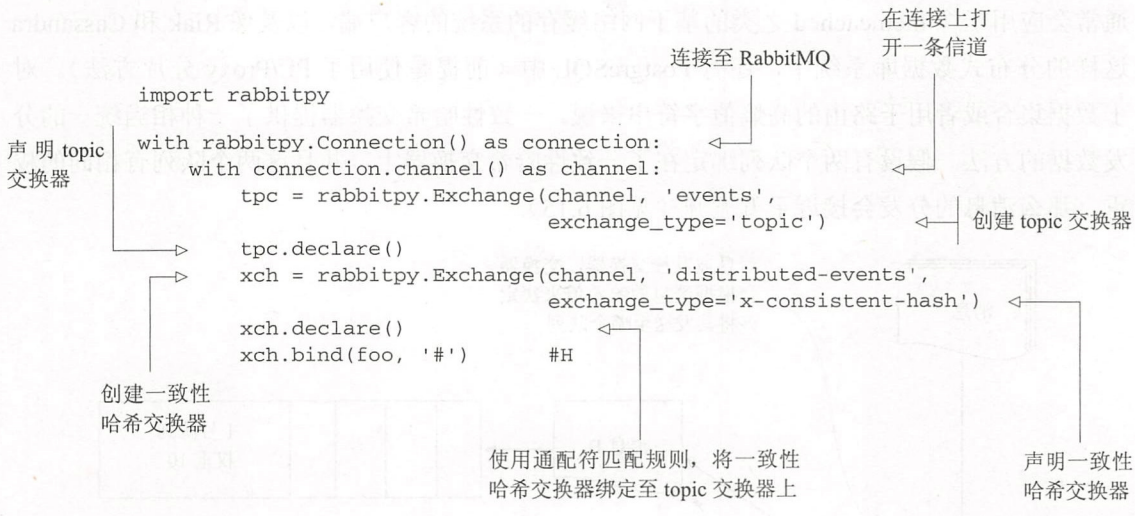


图 6.14　这个小型示例展示了交换器间绑定带来的灵活性

在下列来自“6.6 Exchange Binding”笔记文件中的示例程序中，名为 `distributed-events` 的一致性哈希交换器绑定到了一个名为 events 的 topic 交换器上，将以 any 路由键路由过来的消息分发到绑定在一致性哈希交换器的队列中。



作为一种工具，交换器间绑定为消息通信模式带来了巨大的灵活性。但是这一灵活性也伴随着额外的复杂性和开销。在超级复杂的路由器键绑定模式把人逼疯之前，请记住当发生问题的时候，简单的架构更容易维护和诊断。如果你正考虑使用交换器间绑定，那么你应当确保有相应的功能用例来应对额外的复杂性和开销。

6.7　使用一致性哈希交换器路由消息

一致性哈希交换器（consistent-hashing exchange）插件随 RabbitMQ 一同发布，它将数据分发给绑定的队列上。它可以为用于接收消息的队列做负载均衡。你可以在集群中用该插件来将消息分发到不同物理服务器上的队列中去，或者分发到那些只有单个消费者的队列中去。相比 RabbitMQ 将消息分发至单个队列的多个消费者来说，它提供了潜在的更为快速的吞吐量。当使用数据库或者其他系统以消费者的身份直接集成至 RabbitMQ 上时，一致性哈希交换器提供了扩展数据的能力而无须编写中间件。

注意　如果你考虑使用一致性哈希交换器来提升消费者吞吐量的话，你应当为以下两种场景建立基准测试并加以区分：单队列多消费者和仅有单个消费者的多个队列。这样一来就能辨别出哪一种选择更合适。

一致性哈希交换器采用一致性哈希算法来决定哪个队列将会收到消息。所有队列都有可能成为消息潜在的目的地。不同于将队列绑定至路由键或者头信息，队列将被绑定至一个基于整型的权重值。算法的一部分将采用该权重值来决定消息应如何投递。一致性哈希算法

通常会应用于像 memcached 之类的基于网络缓存的系统的客户端，以及像 Riak 和 Cassandra 这样的分布式数据库系统中，还有 PostgreSQL 中（前提是使用了 PL/Proxy 分片方法）。对于数据集合或者用于路由的高熵值字符串来说，一致性哈希交换器提供了一种相当统一的分发数据的方法。假设有两个队列绑定在了一致性哈希交换器上，并且这两个队列有相同的权重，那么消息的分发会接近于五五开（见图 6.15）。

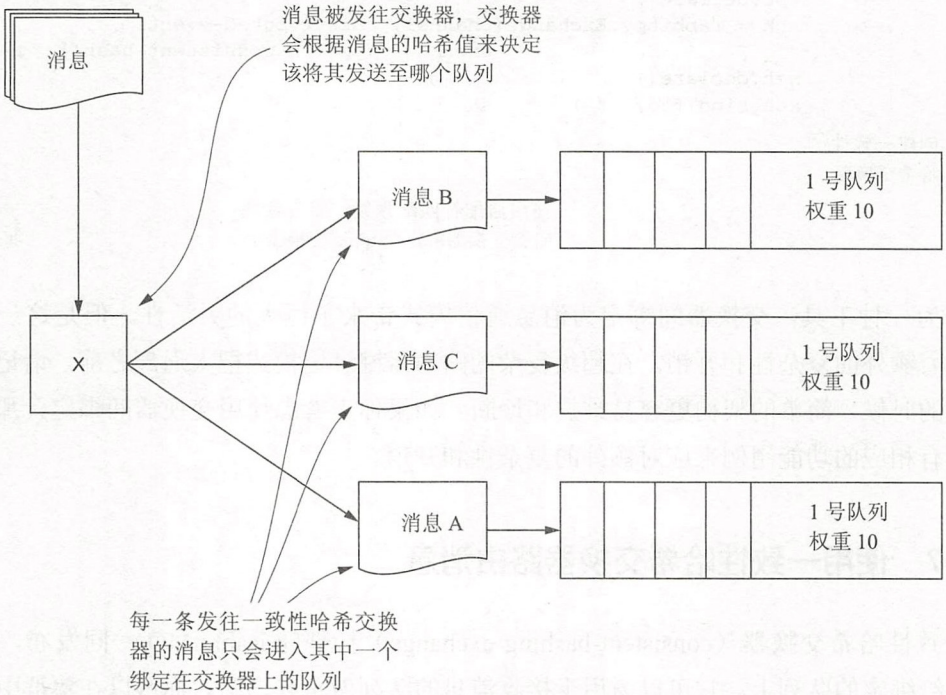


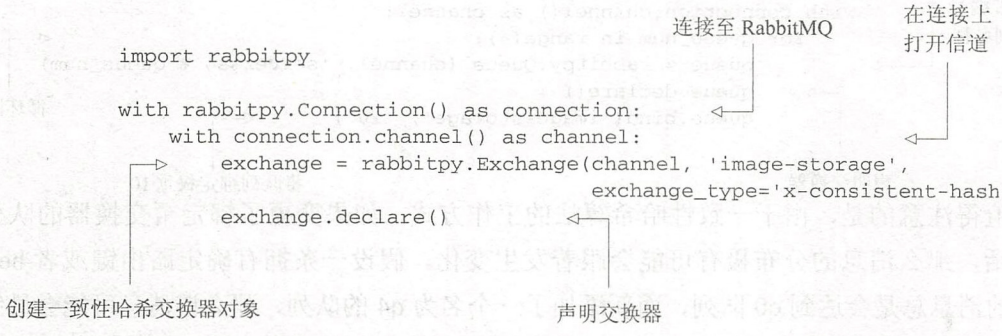
图 6.15　发往一致性哈希交换器的消息会分发至绑定的队列中

在为消息选择目的地时，你无法施加影响来确保消息的均匀分布。一致性哈希交换器不会轮询（round-robin）消息，而是基于路由键或者消息属性中 header-type 值的哈希值来做出明确的路由。不过，相较而言，拥有更高权重的队列将能从交换器接收到更高比例的消息。当然，消息在多个队列之间的分布依赖于所发送消息拥有不同的路由键或者 header 表值。这些值的不同会给消息的分发带来不确定性。拥有相同路由键的五条消息最终会到达同一个队列。

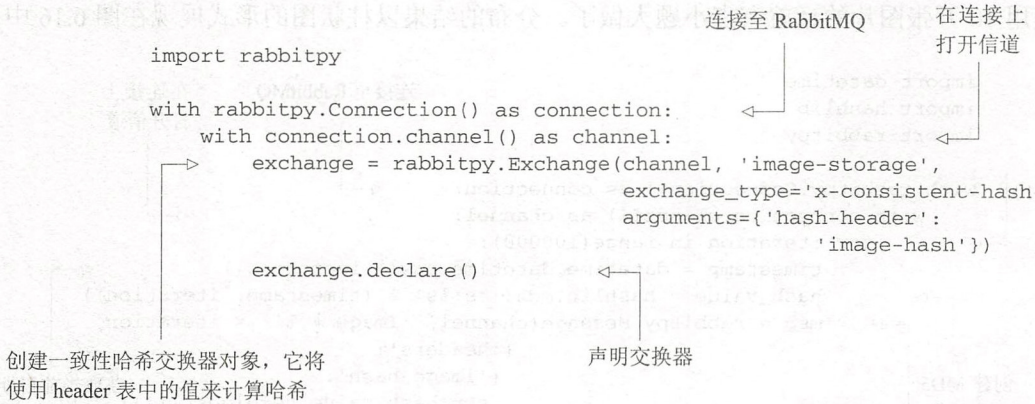
在我们的图像处理 RPC 系统中，这些图像很有可能需要以某种方式进行存储以服务于其他 HTTP 客户端。在处理存储的可扩展性需求时，使用分布式存储解决方案是很常见的需求。在接下来的示例中，我们将采用一致性哈希交换器来将消息分布到四个队列，以便将图

片存储到四个不同的存储服务器中。

默认情况下将采用路由键的哈希值来分发消息。对于图片来说，我们将采用图片本身的哈希值作为路由键。哈希计算方法类似于“6.2.2 Hashing Consumer”笔记文件中所产生的哈希。如果你打算通过路由键值的哈希来分发消息的话，那么在声明交换器时不需要什么特别的设置。上述内容在“6.7 A Consistent-Hashing Exchange that Routes on a Routing Key”笔记文件中。

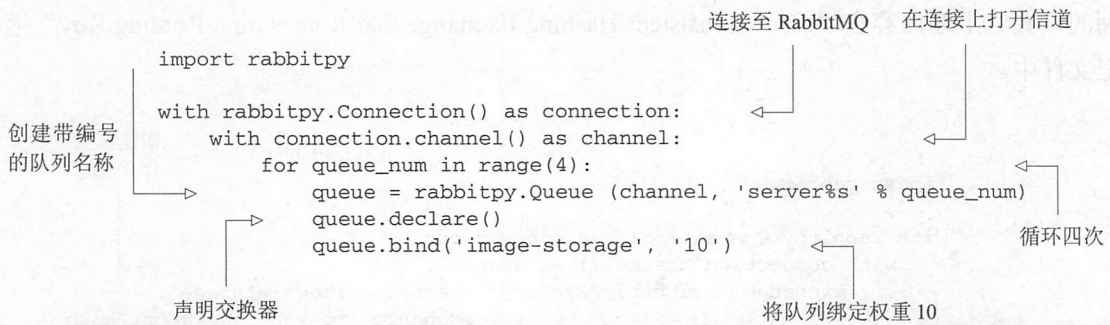


另一种方案是以 header 属性表中的值作为哈希值。当采用该方案时，必须在声明交换器时传入 hash-header 值。hash-header 值包含了 headers 表中的单一键，该键所包含的值将用于消息的哈希计算。上述内容在下列来自“6.7 A Consistent-Hashing Exchange that Routes on a Header”笔记文件的代码片段。



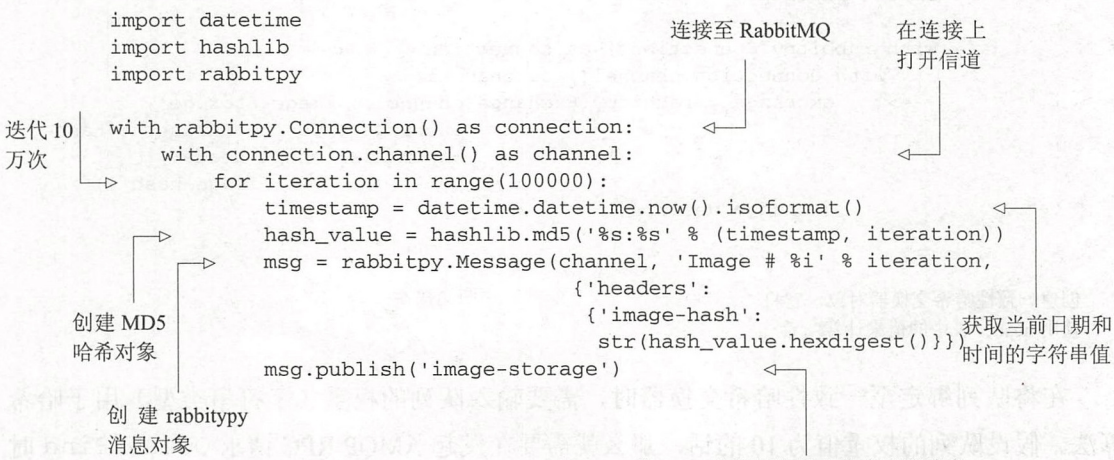
在将队列绑定至一致性哈希交换器时，需要输入队列的权重（字符串类型）用于哈希算法。假设队列的权重值为 10 的话，那么就需要在发起 AMQP RPC 请求 Queue.Bind 时传入字符串值 10 作为绑定键。在我们的图像存储示例中，假设用于存储图像的服务器拥有

不同的存储容量。你需要为较大容量的服务器设定更高的权重值。你甚至可以将权重值指定为对应服务器容量的 GB 值或者 TB 值，以尽可能地实现均衡分布。下列代码示例来自“6.7 Creating `Multiple` Bound Queues”笔记文件中，它将创建四个队列，分别命名为 q0、q1、q2 和 q3，并将它们同等地绑定至 `image-storage` 交换器。



值得注意的是，由于一致性哈希算法的工作方式，如果变更了绑定至交换器的队列总数的话，那么消息的分布极有可能会跟着发生变化。假设一条拥有确定路由键或者 header 表值的消息总是会达到 q0 队列，而新添加了一个名为 q4 的队列，那么该消息可能会最终会达到 5 个队列中的任何一个。之后，拥有相同路由键的消息会始终到达该队列中，直到队列的总数再次发生变化。

为了更好地展示采用一致性哈希交换器时数据的分布情况，下列来自“6.7 Simulated Image Publisher”的代码，向 `image-storage` 交换器发送了 10 万条消息。其中，路由键是用当前时间和消息编号组合起来计算得出的 MD5 哈希值。这样做的原因是因为在本示例中使用 10 万张图片的话就有点小题大做了。分布的结果以柱状图的形式展现在图 6.16 中。



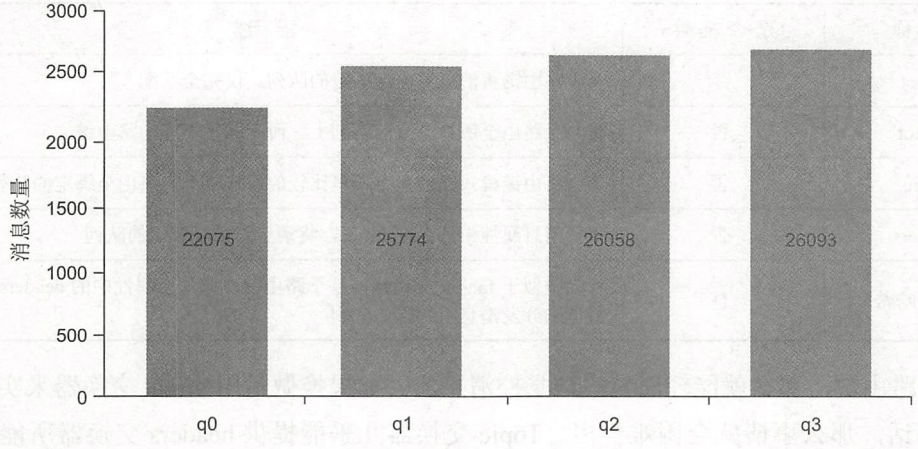


图 6.16　公平随机哈希下的 10 万条消息的分布情况

正如你所见到的那样，消息分布接近均衡但并非完全一致。这是因为投递至哪个队列是由用于路由的值决定的。除非对具体路由键的值精心设计，否则无法像轮询（round-robin）的方式那样真正地做到消息的负载均衡。如果你想要在多个队列之间对消息进行负载均衡，而又不想使用一致性哈希方法的话，那么请参考 John Brisbin 的随机交换器（https://github.com/jbrisbin/random-exchange）吧。这种交换器并不使用路由键来将消息分发至队列当中去，而是使用随机数字生成的方式。RabbitMQ 的插件灵活多变，将来有一天诞生真正意义上的轮询交换器也不足为奇。如果这正好激发了你的兴趣，那么就放手去做吧。

如果你期望利用一致性哈希交换器来增加吞吐量的话，那么且慢，它通常不会提升性能或者消息吞吐量。但是如果你想要实现跨越数据中心或者 RabbitMQ 集群来分发部分消息的话，那么一致性哈希交换器将能体现其价值。

6.8　小结

到目前为止，你应当对 RabbitMQ 内置的多种路由机制有了充分的理解。表 6.1 罗列并总结了交换器以及对应的描述，以供参考。每种交换器类型提供了独特的功能，应用程序可以充分利用它们来确保消息能够尽快地路由到合适的消费者那里。

记住，在架构创建的初期你无法预测消息将以怎样的方式被重用。因此，我建议在构建消息通信架构时要尽可能的灵活。在使用 topic 交换器时，采用命名空间的方式来定义富

表 6.1 交换器类型总结

名 称	是否插件	描 述
Direct	否	基于路由键将消息路由至绑定的队列。仅完全匹配
Fanout	否	将消息路由至所有绑定的队列上，而无关乎消息的路由键
Topic	否	使用路由键模式匹配和字符串比较的方式将消息路由至绑定的队列
Headers	否	基于消息属性中的 headers 表，将消息路由至绑定的队列
一致性哈希	是	行为类似于 fanout 交换器，基于路由键或者消息属性中的 headers 表的哈希值来分发消息至绑定的队列

含语义的路由键，那么就能轻而易举地掌控消息流。如果换做是用 direct 交换器来实现主路由机制的话，那么事情就会困难得多。Topic 交换器几乎能提供 headers 交换器所能提供的相同级别的灵活性，而不会因为用到了 AMQP 消息属性而和协议绑死。

从内部原理来讲，对于流向 RabbitMQ 的消息来说交换器就是简单的消息路由机制。还有各种各样的交换器插件，例如用于将消息存储至数据库的交换器，例如 Riak 交换器（https://github.com/jbrisbin/ riak-exchange），以及内存型交换器，例如 Message History 交换器（https://github.com/videlalvaro/rabbitmq-recent-history-exchange）。

在第 7 章中，你将学习如何将多个 RabbitMQ 服务器组合成内聚消息通信集群，这不仅为消息通信吞吐量的扩展提供了方法，同时通过采用高可用队列为消息通信提供了强有力的保障。

第二篇

管理数据中心或云中的 RabbitMQ

在应用程序开发的早期阶段引入 RabbitMQ 对应用程序的架构大有裨益。作为开发者，我们不能将代码丢给产品基础架构团队就甩手了事，应当担负起理解基础架构设置的职责。本书的第二篇内容主要讲解 RabbitMQ 的集群：集群的设置、集群的运作方式以及集群的管理。我们将看到消息在网络环境下如何分发和复制：我们将利用联合交换器和队列，处理跨越多个物理上分隔的集群，并在这些集群间进行复制。

第 7 章　RabbitMQ 集群

本章概要：

- 集群管理
- 队列位置如何影响性能
- 集群设置步骤
- 节点崩溃时的应对方法

RabbitMQ 作为消息代理服务器，对于独立应用来说是简直完美。但当应用程序需要高可用队列来满足额外的投递保证，或者将 RabbitMQ 用作众多应用程序的中心消息通信总线时，RabbitMQ 内建的集群能力能够跨越多台服务器，提供强大的、内聚的环境。

笔者将从 RabbitMQ 集群的功能和行为描述开始，接下来我们将在 Vagrant 虚拟机（VM）环境下配置一个双节点的 RabbitMQ 集群。此外，你将学到为什么对于高效的集群来说队列的位置至关重要，以及如何配置 HA 队列。还有，你将学习 RabbitMQ 集群的底层工作机制，哪些服务器资源对于确保集群的性能和稳定性来说是最重要的。在本章的最后，你将学到如何从崩溃和节点故障中恢复。

7.1 集群简介

RabbitMQ 集群无缝封装了多台 RabbitMQ 服务器。在 RabbitMQ 集群里，运行时状态包含交换器、队列、绑定器、用户、虚拟主机以及策略，它们对所有节点都可用。由于这种共享运行时状态的特性，集群中的每个节点都能绑定、发布或者删除连接到第一个节点时创建的交换器（见图 7.1）。

RabbitMQ 内建的集群机制提供了一种激动人心的方式来扩展 RabbitMQ。此外，RabbitMQ 集群为消息的发布者和消费者提供了创建结构化架构的机制。在大型集群环境中，有专门的节点来服务特定的任务或者队列，这一点不足为奇。举例来说，一些集群节点专门用于服务前端应用程序发布消息，而另一些节点则只用来服务队列和消费者。集群提供了绝佳的方式来创建 HA（高可用）队列，以创建可以容错的 RabbitMQ 环境。HA 队列可以跨越多个集群节点并共享同步队列状态和消息数据。假设 HA 队列中的某个节点发生故障的话，集群中的其他节点仍然保存着消息和队列状态。当故障的节点重新加入集群时，该节点会完全同步自节点故障以来所有被消费的消息。

虽然 RabbitMQ 内建集群有不少优点，但是对其本身的限制和劣势的认识也很重要。首先，集群是按照低延迟环境进行设计的。千万不要跨越 WAN 或者互联网来搭建 RabbitMQ 集群。集群中的状态同步和跨节点的消息投递需要低延迟的通信，只有 LAN 可以满足这一要求。你可以在诸如 Amazon EC2 这样的云端环境中运行 RabbitMQ，但注意在使用时不要跨越可用区。为了在高延迟环境下同步 RabbitMQ 消息，你可以参考第 8 章介绍的 Shovel 和 Federation 工具。

另一个需要重点考量的问题是 RabbitMQ 的集群大小。维护集群共享状态的工作和开销与集群中节点数量的多少成正比。举例来说，使用管理 API 来获取大型集群中的统计数据

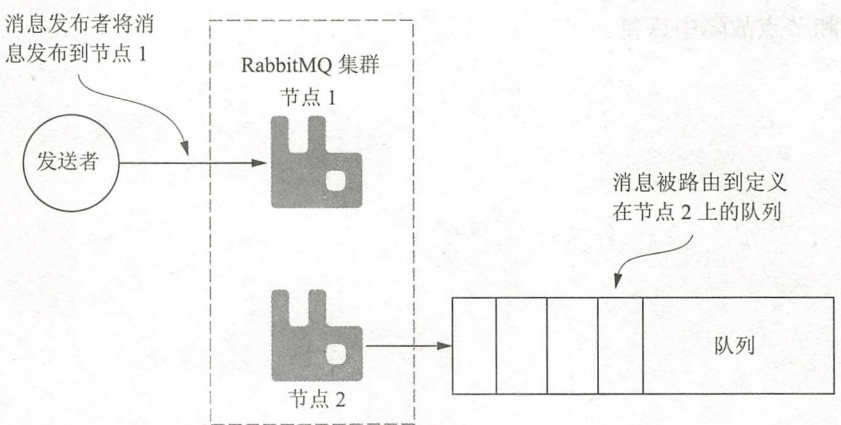


图 7.1　集群中跨节点的消息投递

所需要的时间就要比单个节点的要长得多。上述操作耗费的时间和集群中最慢的那个节点响应速度有关。RabbitMQ 社区中的传统观念要求集群中节点数量的上限在 32 到 64 个。记住，每向集群中新添加一个节点，你就为集群中的状态同步增加了更多的复杂性。集群中的每个节点必须知道集群中其他节点。这种非线性的复杂度会拖慢跨节点的消息投递和集群的管理。幸运的是，即使面对这一复杂问题，RabbitMQ 管理界面（management UI）考虑到了这一复杂性，它能够支持管理大型集群。

7.1.1　集群和管理界面

在 RabbitMQ 管理界面上管理集群的方式和管理单个节点的那样如出一辙（见图 7.2）。这有助于你对 RabbitMQ 集群的理解。管理界面上的 Overview 页包含了 RabbitMQ 集群和集群中节点的顶层信息。

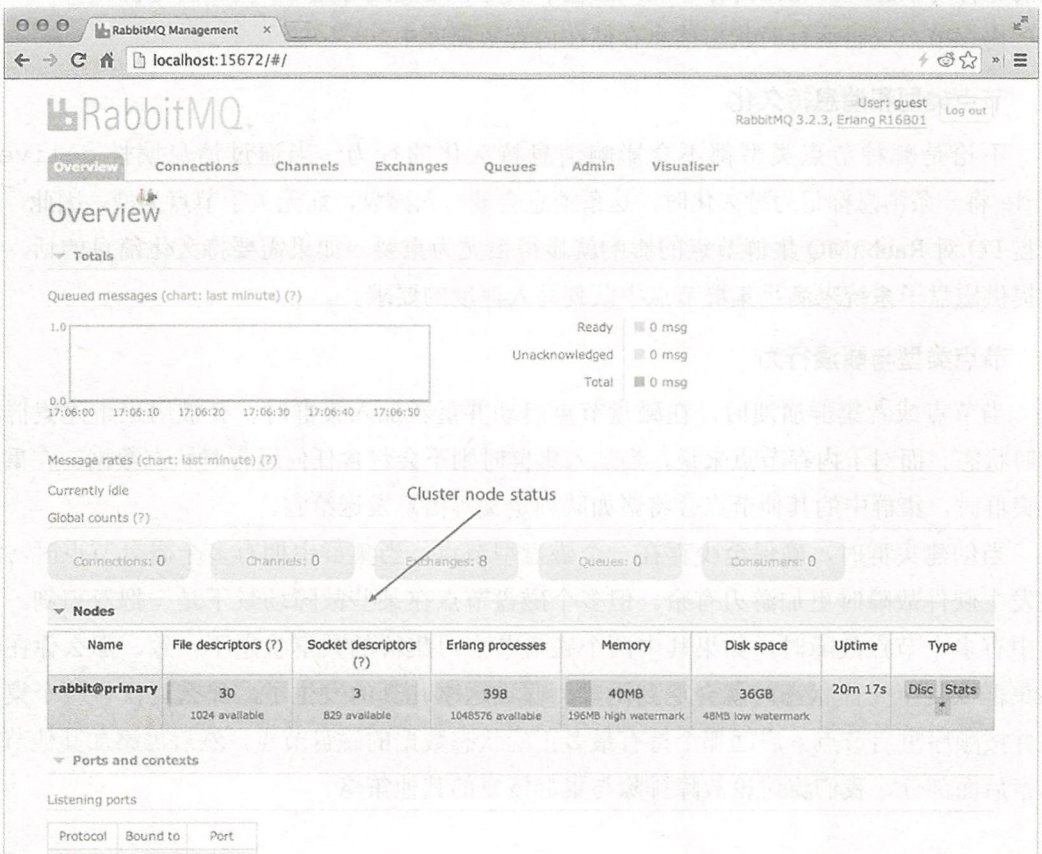


图 7.2　管理界面使用单个节点来突出表示集群状态

在图 7.2 的突出表示区域中，集群节点以列表的形式展现，每一列都描述了总体健康度和状态。当向集群中添加节点时，这些节点会添加到表格中。在更大型的集群中，该表格会花费更多的时间来刷新。这是因为每次调用 API 来收集信息时，集群中的每个节点都要优先响应更新信息的查询。

在更深入了解集群的管理界面之前，让我们先来了解 RabbitMQ 集群中的节点类型。

7.1.2 集群节点类型

RabbitMQ集群有多种不同类型的节点，它们的行为都不太一样。当向集群中添加节点时，它必须是磁盘节点或者内存节点。

磁盘节点将集群的运行时状态会同时存储在内存和磁盘上。在 RabbitMQ 中，运行时状态包括集群、队列、绑定、虚拟主机、用户和策略等信息的定义。鉴于此，如果集群拥有大量的运行时状态时，相比内存节点，磁盘节点更容易受到磁盘 I/O 问题的困扰。

内存节点仅将运行时状态信息存储在内存数据库中。

节点类型和消息持久化

不论是哪种节点类型都不会影响消息持久化的行为。当通过消息属性 `delivery-mode` 将一条消息标记为持久化时，这条消息会被写入磁盘，而无关乎节点类型。因此，考虑磁盘 I/O 对 RabbitMQ 集群节点的影响就显得很尤为重要。如果需要持久化消息的话，你应当提供磁盘子系统来满足集群节点中队列写入速度的要求。

节点类型与崩溃行为

当节点或者集群崩溃时，在磁盘节点启动并重新加入集群时，会被用来重建集群的运行时状态。而对于内存节点来说，当加入集群时则不会包含任何运行时状态数据。在重新加入集群时，集群中的其他节点会将诸如队列定义等信息发送给它。

当创建集群时，确保至少存在一个磁盘型节点。当集群中拥有多个磁盘节点时，就能在发生硬件故障时更加游刃有余。但多个磁盘节点在某些故障场景下是一把双刃剑。当集群中有多个节点故障时，如果其中两个磁盘节点对集群的共享状态不一致，那么你在尝试将集群恢复至先前状态时就会遇到困难。假设这事儿真的发生了，那么建议将整个集群关闭并按顺序重启节点。启动那个持有最多正确状态数据的磁盘节点，然后再添加其他节点。本章后面部分，我们将讨论故障排除与集群恢复的其他策略。

状态节点

如果你使用 rabbitmq 管理插件的话，那么你可以使用另一种节点类型，即统计节点（Stats node）。它只能和磁盘节点搭配使用。统计节点负责收集集群中每个节点的全部统计数据和状态数据。在任意时刻，一个集群只能有一个统计节点。对于大型集群设置来说，最佳策略是配置专门的管理节点，即主磁盘节点和统计节点，并再至少配置一个磁盘节点以提供故障转移的能力（见图 7.3）。

根据管理 API 的使用频率和用途以及 RabbitMQ 中使用的资源数量，运行管理 API 可能会带来较高的 CPU 成本。运行专们用于管理的节点服务器可确保消息投递和统计数据收集这两之间互不影响。

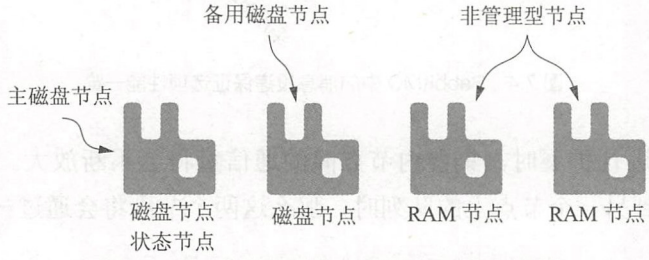


图 7.3　带有备用磁盘节点和两个内存节点的集群

在拥有两个磁盘节点的集群拓扑设置中，如果主节点发生故障的话，统计节点将会被指派给备用磁盘节点。当主磁盘节点恢复并重新加入集群后，它并不会重新获得统计节点，除非被指派为统计节点的备用节点停止运行或者离开集群。

在 RabbitMQ 集群的管理方面统计节点扮演着重要的角色。要是没有了 RabbitMQ 管理插件和统计节点的话，获取集群范围内的性能、连接、队列深度和运营问题等数据将变得十分困难。

7.1.3　集群和队列行为

当消息发送到集群中的任何一个节点时，该消息会被路由到队列中去，无关乎队列在集群中的位置。当声明一个队列时，该队列会在 RPC 请求 `Queue.Declare` 发送的集群节点上创建。选择在哪个节点上声明队列将对消息的吞吐量和性能造成影响。如果节点拥有过多的队列、消息发布者和消费者，那么它要比集群中三者（队列、发布者和消费者）均匀分布的情况下要慢。除了资源利用的不平均分布外，在集群中随意创建队列会对消息的发布和消费造成影响。

消息发布的考量

你可能对图 7.4 有印象，这是由第 4 章中的图稍作修改而来的。当向集群发送消息时，这时性能度量比单个 RabbitMQ 服务器显得更为重要。

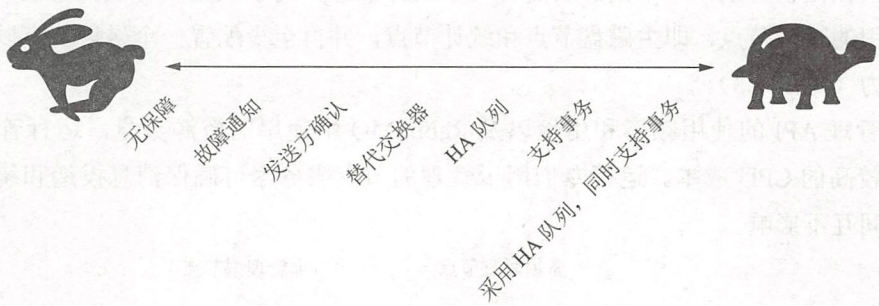


图 7.4　RabbitMQ 中的消息投递保证选项性能一览

当你从左到右进行扩展时，集群内节点间的通信量将会不断放大。当你向一个节点发送的消息将被路由到另一个节点上的队列时，那么这两个节点将会通过一种消息投递保证的方法来进行协调。

举例来说，图 7.5 展示了在发布者保证机制中，发布的消息在节点间的逻辑步骤。

虽然图 7.5 列举的步骤并不会大大减少消息的吞吐量，但是在采用集群的方式构建消息通信架构时应当考虑到这种确认行为所带来的复杂性。通过各种节点上的发布者和消费者对

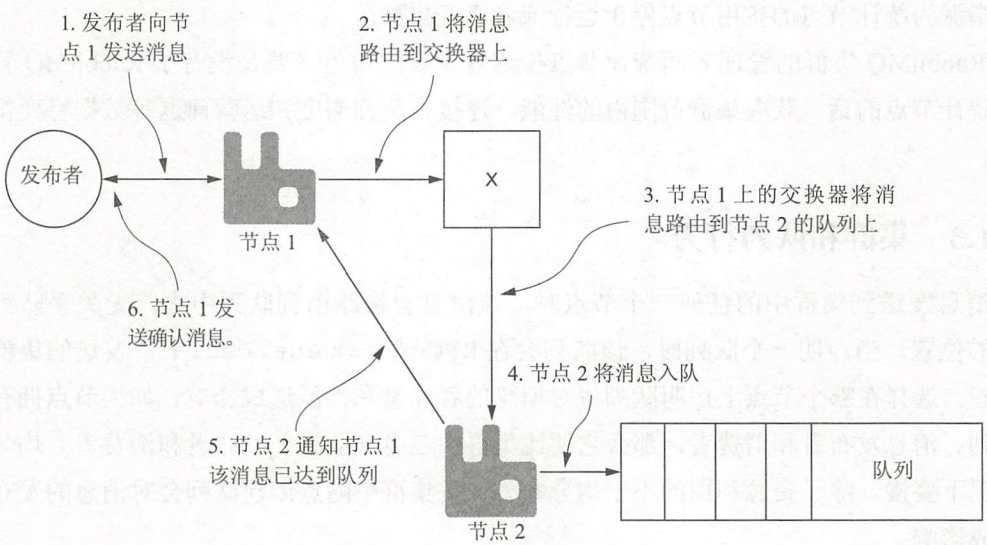


图 7.5　具有消费保证的多点发布

各种方法进行基准测试，可以从中找到最适合的方案。吞吐量可能并不是成功实现消息通信架构的最优指标。当然，性能不佳会带来消极影响。

对于单个节点来说，消息发布仅仅是消息吞吐量的其中一方面。集群也会对消费者的消息吞吐量造成影响。

特定节点的消费者

为了提升集群中的消息吞吐量，RabbitMQ 会尽可能尝试将新发来的消息路由到预先存在的消费者去。但是，当队列产生消息堆积时，新消息将会被发布到集群中队列定义的节点。在这种场景下，如果你将消费者连上了没有定义该队列的其他节点的话，性能将受到影响（见图 7.6）。

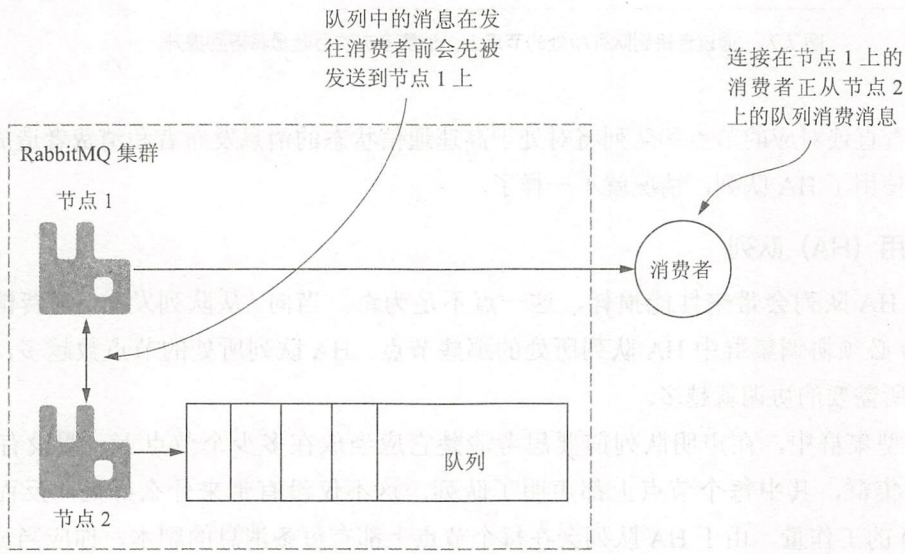


图 7.6　跨集群节点消费消息

在这一场景中，消息被发送到节点 2 上的队列中，而消费者则位于节点 1。在连接在节点 1 上的消费者获取消息时，这些消息必须先通过集群路由到节点 1 才行。如果在连接消费者之前考虑到了队列所处的位置，那就能减少上述开销。相比之前所述的场景，队列所在的节点可以直接将消息投递给连接着的消费者（见图 7.7）。

队列、消费者、发发布者各就其位的好处是能减少集群内的通信，并提高总体消息吞

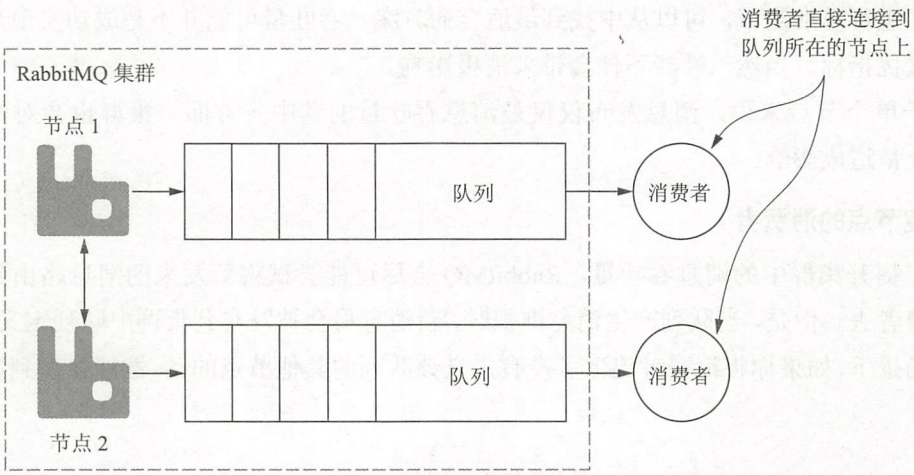


图 7.7　通过连接到队列所处的节点上，消费者端的吞吐量将得到提升

吐量。是否直连对应的节点和队列将对处于高速通信状态的消息发布者和消费者造成巨大的影响。若使用了 HA 队列，情况就不一样了。

高可用（HA）队列

使用 HA 队列会带来性能损耗，这一点不足为奇。当向 / 从队列发布 / 消费消息时，RabbitMQ 必须协调集群中 HA 队列所处的那些节点。HA 队列所处的节点数越多，那么这些节点间所需要的协调就越多。

在大型集群中，在声明队列前要思考清楚它应当放在多少个节点上。假设有一个 24 个节点的集群，其中每个节点上都声明了队列，这不仅没有带来什么好处，反而加重了 RabbitMQ 的工作量。由于 HA 队列会在每个节点上都有每条消息的副本，你应当反复斟酌到底是否需要超过 2 个或者 3 个节点来确保消息不会丢失。

7.2　集群设置

RabbitMQ 集群至少需要两个节点。在本节中，你将使用 Vagrant VM 来设置集群。从附录中下载的 Vagrant 配置（我们在第 2 章中设置过）已经针对后续的示例程序做配置。主 VM 将用作集群的第一台服务器。你将在本章配置第二台使用 Vagrant 配置中的 VM。

启动并登录第二台 VM，开始着手设置 RabbitMQ 集群吧。

7.2.1　虚拟机设置

先将目录切换到之前为本书配置环境时解压 rmqid-vagrant.zip 文件的位置。通过输入下列指令来启动第二台 VM：

```
vagrant up secondary
```

这一操作会启动第二台 VM，你将使用它来进行 RabbitMQ 集群的实验。当 VM 设置完成时，你将看到类似图 7.8 的输出。

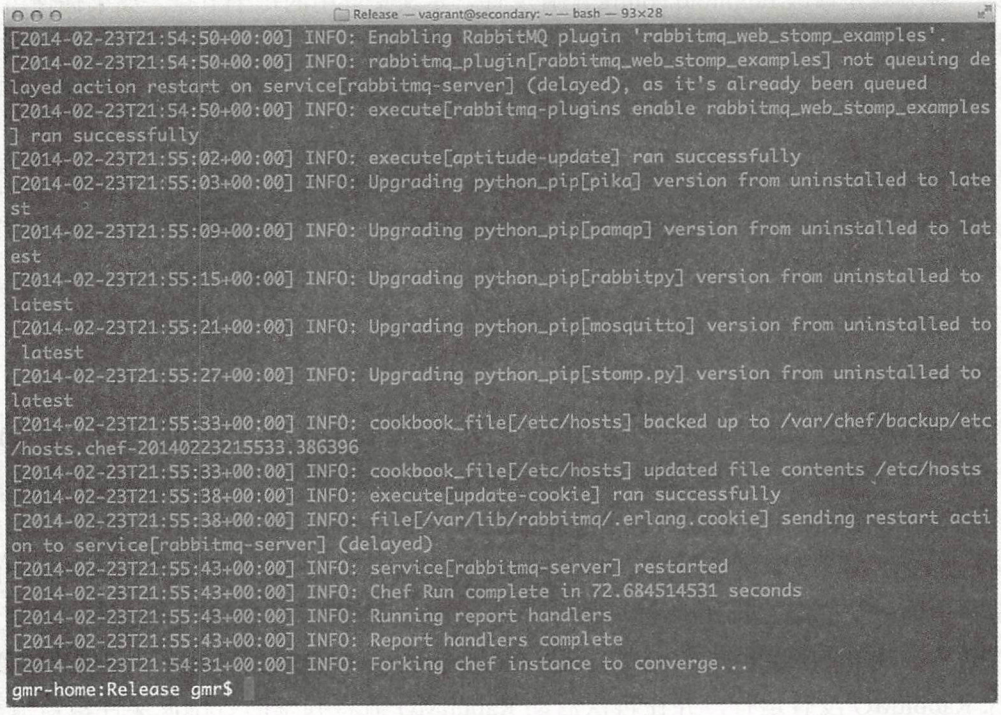


图 7.8　第二台 vagrant 启动输出

VM 启动运行之后，你可以在同一目录下通过运行下列 Vagrant 命令来启动 ssh。

```
vagrant ssh secondary
```

现在，你应当已经作为 vagrant 用户连接上第二台 VM 了。当然，如果你需要以 root 用户身份运行命令的话，应当使用下列命令来切换到 root 用户：

```
sudo su -
```

当运行上述命令后，你应当看到从 vagrant@secondary:~$ 到 root@secondary:~# 的提示。这代表你将以 VM 的 root 用户身份登录。作为 root 用户你将有权限来运行 rabbit-mqctl 脚本来和本机上的 RabbitMQ 服务器实例进行通信。

现在让我们来设置集群吧。

7.2.2 向集群中添加节点

有两种方式向 RabbitMQ 集群中添加节点。

第一种方式是修改 rabbitmq.config 配置文件，定义集群中的每个节点。如果你使用自动配置管理工具，例如 Chef (www.getchef.com) 或者 Puppet (www.puppet-labs.com)，并且在一开始就定义好集群的话，那么这种方式更为推荐。在通过 rabbitmq.config 文件创建集群前，先手工创建一个节点。

另一种方式是通过使用 rabbitmqctl 命令行工具来向集群中添加或删除节点。这种方式提供了相对并不严格的方式来学习 RabbitMQ 集群行为，有助于了解集群降级问题的排查。在本节中，你将利用 rabbitmqctl 来为这两台 VM 创建集群。但在此之前，你应当了解下 Erlang cookies，及其对 RabbitMQ 集群的作用。

ERLANG COOKIES

RabbitMQ 使用 Erlang 内建的多节点通信机制来实现节点间的通信。Erlang 和 RabbitMQ 进程使用共享的安全文件 *cookie*。Erlang cookie 文件包含在 RabbitMQ 的数据目录下。在 *NIX 平台上，该文件通常位于 /var/lib/rabbitmq/.erlang.cookie 中，不同的版本间会有所差异。cookie 文件包含了一个短字符串。集群中的每个节点的这个短字符串都是相同的。如果集群中的每个节点上的 cookie 文件都不一样的话，那么它们之间就不能相互通信了。

在首次启动 RabbitMQ 时或者该文件丢失时，cookie 文件会被创建出来。在设置集群时，请确保 RabbitMQ 没有运行，并在再次启动 RabbitMQ 前将共享的 cookie 文件覆盖原有的那个 cookie 文件。本书自带的 Vagrant VM 配置手册中已经在两台机器上都设置好了 Erlang cookie。也就是说，你可以开始使用 rabbitmqctl 来创建集群了。

注意 使用 rabbitmqctl 是添加和删除集群节点的简单方法。我们也可以用它来将节点在磁盘节点和内存节点间来回切换。rabbitmqctl 本质上包装了 Erlang 应用程序。该程序负责和 RabbitMQ 通信。正因如此，它需要访问 Erlang cookie。当以 root 用户身份运行命令时，rabbitmqctl 知道如何找到并使用 cookie 文件。如果你在产品环境中使用 rabbitmqctl 时遇到问题，请确保你运行 rabbitmqctl 的用户拥有访问 RabbitMQ Erlang cookie 的权限，或者在 home 目录下存有该文件的副本。

创建点对点集群

首先在节点上运行 RabbitMQ，同时以 root 用户身份进行登录。现在开始添加第二个 VM 节点，连同主 VM 节点一起创建集群。

为了实现上述目标，首先你需要使用 `rabbitmqctl` 来停止第二个节点上的 RabbitMQ。你不会停止 RabbitMQ 服务器进程本身，而是使用 `rabbitmq` 来指示 RabbitMQ 停止 Erlang 中的内部进程。在终端键入下列命令：

```
rabbitmqctl stop_app
```

你应当能看到类似下列的输出：

```
Stopping node rabbit@secondary ...
...done.
```

在进程停止运行之后，你需要清除 RabbitMQ 节点的状态，包括任何运行时配置数据和状态。为了达到这一目标，你需要键入下列命令来重置它的内部数据库：

```
rabbitmqctl reset
```

你应当能看到如下类似的响应：

```
Resetting node rabbit@secondary ...
...done.
```

现在你可以将它加入主节点构成集群了：

```
rabbitmqctl join_cluster rabbit@primary
```

你应当能看到如下输出：

```
Clustering node rabbit@secondary with rabbit@primary ...
...done.
```

最后，使用下列命令再次启动服务器：

```
rabbitmqctl start_app
```

你应当能看到如下输出：

```
Starting node rabbit@secondary ...
...done.
```

恭喜！包含两个节点的 RabbitMQ 集群已经搭建完成。打开 http://localhost:15672，你可以在浏览器管理界面上看到如图 7.9 所示的 Overview 页面。

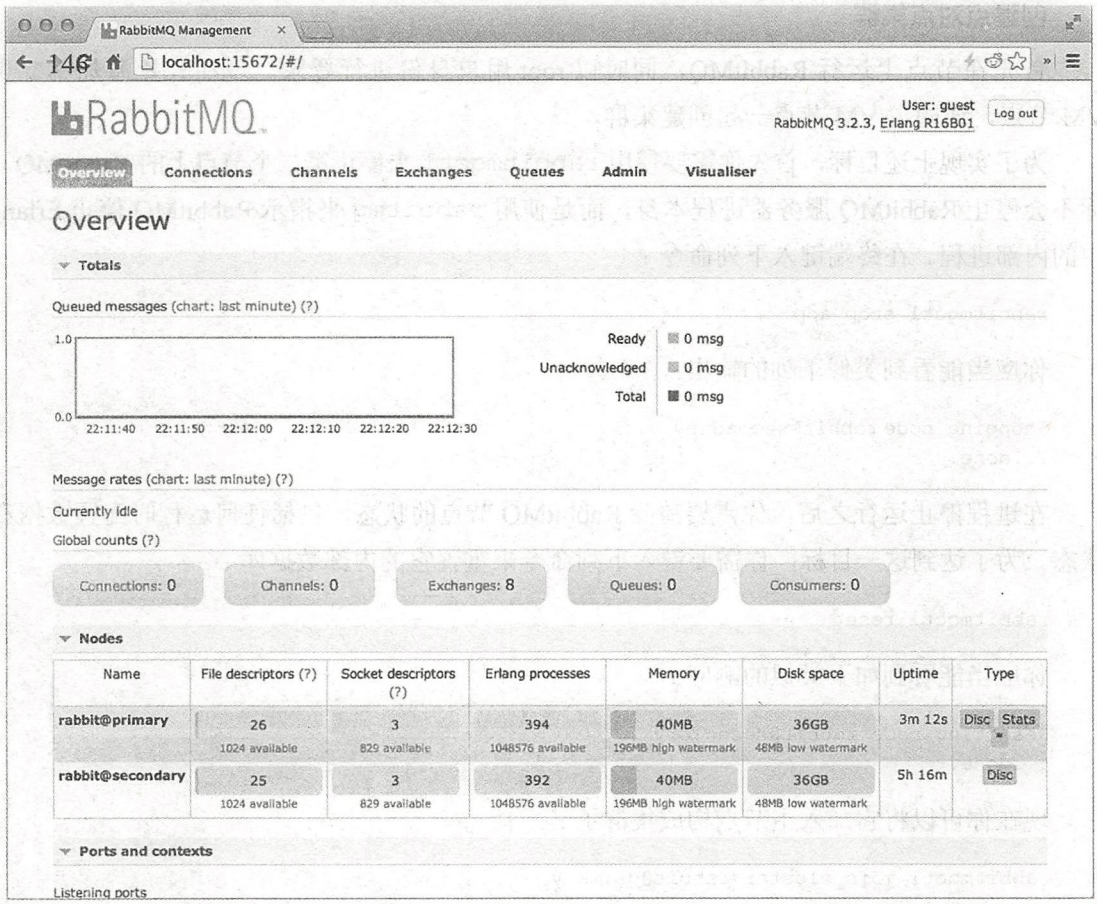


图 7.9　拥有两个节点的 RabbitMQ 集群

基于配置的集群

使用配置文件来创建集群比较复杂。当使用 `rabbitmqctl` 来设置集群时，可以发送 `reset` 命令给服务器，服务器会清除所有状态和内部数据。但是在使用基于文件的集群配置时就不能这么做了，因为 RabbitMQ 会在服务器启动的时候尝试将节点加入到集群中。如果你安装了 RabbitMQ 并且在创建集群配置文件之前服务器就已经启动的话，节点就无法加入集群了。

如果你使用配置管理工具的话，明智的做法是在安装 RabbitMQ 前就创建好 /etc/rabbitmq.config 文件。新安装的程序不应该覆盖先前就已存在的配置文件。推荐在此配置阶段就编写好整个集群节点共享的 Erlang cookie 文件。

在配置文件中定义集群看起来直截了当。在 /etc/rabbitmq.config 文件中，cluster_nodes 定义了集群中的节点列表，以及每个节点是磁盘节点还是内存节点。以下配置片段就是用来定义之前创建的 VM 集群的：

```
[{rabbit,
  [{cluster_nodes, {['rabbit@primary', 'rabbit@secondary'],disc}}]
}].
```

如果在两个节点上使用这一套配置，那么这两者都将被设定为磁盘节点。如果想要将备用节点设置为内存节点的话，你需要修改配置，将 disc 关键字替换为 ram：

```
[{rabbit,
  [{cluster_nodes, {['rabbit@primary', 'rabbit@secondary'], ram}}]
}].
```

基于配置文件的集群会带来一些不便。由于节点的定义存放在了配置文件中，因此添加和删除节点需要更新集群中所有节点的配置文件。同样值得注意的是，集群信息最终会存储为集群中的磁盘节点的状态数据。在配置文件中定义集群使得 RabbitMQ 节点在启动时立刻加入集群。这意味着你对拓扑或者配置的更改是不会对集群中的节点关系造成影响的。

7.3　小结

功能强大的 RabbitMQ 集群能够用来扩展消息通信架构，为消息的发布和消费提供冗余能力。虽然 RabbitMQ 内置的集群拓扑允许从集群中的任一节点发布和消费消息，但为了实现高吞吐量的目标我们必须考虑队列在集群中的位置。

对于局域网环境来说，集群提供了坚固的平台以应对消息通信平台的增长。但是在面对诸如 WAN 和互联网等高延迟的环境时，集群的方式就不能奏效了。我们将在第 8 章介绍两种插件来将分布在 WAN/ 互联网环境中的 RabbitMQ 节点连接起来。

第 8 章　跨集群的消息分发

本章概要：

- 联合交换器和联合队列
- 如何在 AWS 上设置多个联合 RabbitMQ 节点
- RabbitMQ 联合的多种应用模式

当需要实现跨数据中心的消息通信、RabbitMQ 升级以及在不同的 RabbitMQ 集群间提供透明的消息访问能力时，你也许该考虑使用联合插件。联合插件提供了两种不同的方式来实现集群间的消息通信。通过使用联合交换器，发往其中一台 RabbitMQ 服务器或者集群的消息会自动路由到下游主机上已绑定的交换器和队列中去。此外，如果你需要更细化的消息投递方式的话，可以使用联合队列将消息发送至单个队列，而不是交换器。不管哪种方式，插件实现的目标就是要将最初发往上游节点的消息透明地转播到下游节点中去（见图 8.1）。

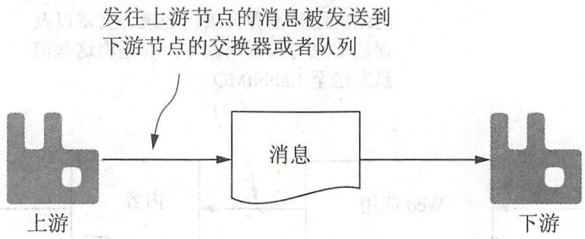


图 8.1　来自上游节点的消息被发送至下游节点的交换器和队列

8.1　联合交换器和联合队列

你的消息通信架构中是否需要采用联合插件？要回答这个问题，需要先理解联合插件的工作机制以及应用联合插件所带来的好处。作为核心 RabbitMQ 发行版的一部分，联合插件提供了灵活的方式透明地实现节点间和集群间的消息中继。这一插件的功能可以分为以下两个主要组件：联合交换器和联合队列。

联合交换器允许发往上游节点交换器的消息被透明地发送至下游节点中相同名称的交换器上。而联合队列则允许下游节点扮演上游节点中共享队列的消费者角色，为多个下游节点提供了轮询（round-robin）消费消息的能力。

在本章的后面部分，你将搭建一个测试环境来试验上述两种类型的联合插件。但首先让我们来了解它们的工作机制。

8.1.1　联合交换器

假设手头上有个任务，需要往云端运行着的 Web 应用程序中添加用户行为相关的大规模数据处理能力。该应用是一个大规模的、用户驱动的新闻站点，类似于 Reddit 或者 Slashdot。该应用已经采用了基于消息的拓扑来满足站点上的用户行为交互。当用户登录、发表文章或者留下评论时，应用并不会直接将这些内容写入数据库，而是将这些消息发送至 RabbitMQ，由消费者来执行数据库的写入操作（见图 8.2）。

由于在应用程序和执行写入数据库操作的消费者之间的通信采用 RabbitMQ 中间件来实现，因此解耦了网站应用程序的数据库写入操作。你可以轻而易举地控制消息流，将数据写入到数据仓库中以备分析之用。其中一种实现方式是将本地消费者添加到 Web 应用程序，以便将数据写入数据仓库。但是如果数据仓库的基础设施和存储位于别处时，该怎么办呢？

就如我们在第 7 章提到的那样，RabbitMQ 的内建集群能力需要低延迟的网络环境，不建议跨网络分区使用。这里的术语网络分区（network partition）指的是网络环境中的节点之间无法通信。

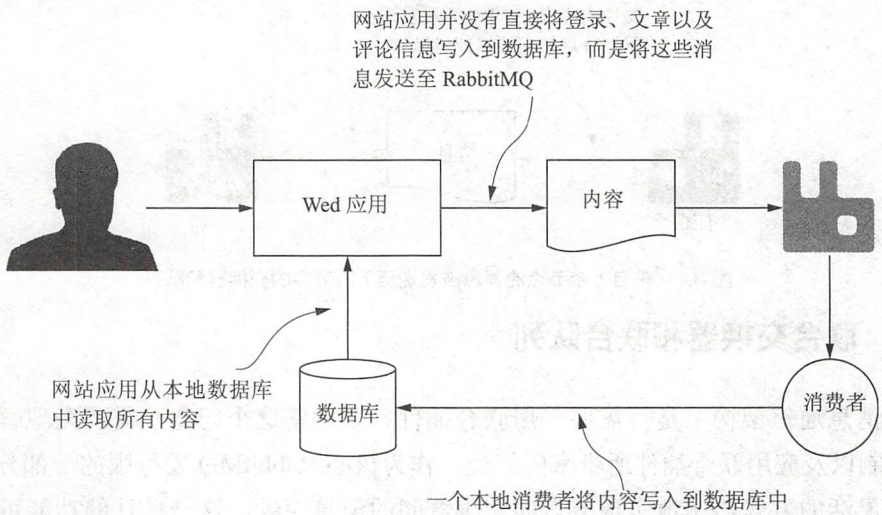


图 8.2　在添加联合插件之前，网站应用已经解耦了写操作

像互联网这种高延迟的网络环境中，网络分区并不罕见，这值得我们认真考虑。幸运的是，RabbitMQ 捆绑的一个插件可以用来在上述情景中将节点连接起来。这个联合插件允许下游 RabbitMQ 服务器可以从先前已经存在的 RabbitMQ 服务器上获取消息（见图 8.3）。

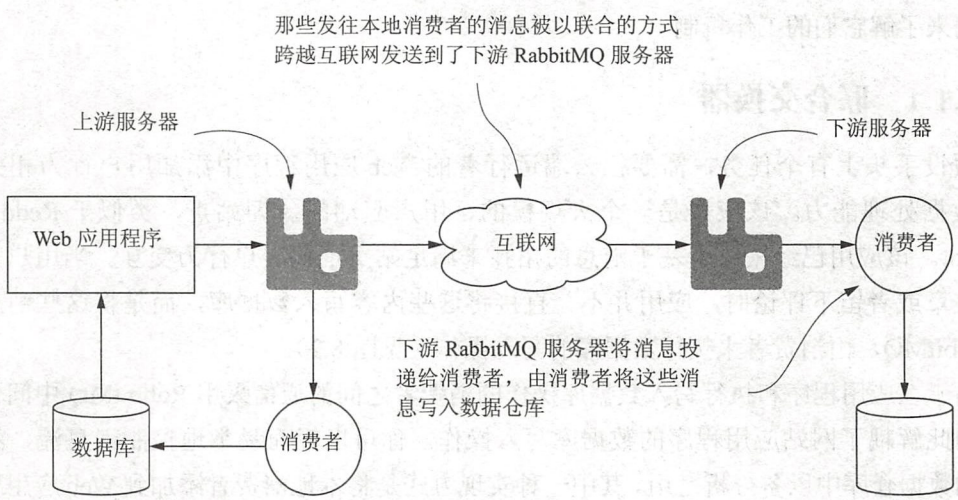


图 8.3　Web 应用程序保持不变，联合了下游 RabbitMQ 服务器，用来将消息存储到数据仓库中

当设定好联合服务器之后，你需要做的只是在消息来源的交换器上创建并应用策略即可。假设上游 RabbitMQ 服务器拥有一个名为 events 的交换器，用来接收发来的登录、文章和评论消息。那么在下游 RabbitMQ 服务器上应当创建一份联合策略，用来匹配该交换器名称。当在下游 RabbitMQ 上创建交换器并绑定了队列时，该策略会让 RabbitMQ 连接到上游服务器，并开始向下游队列发送消息。

在 RabbitMQ 开始从上游服务器向下游队列发送消息后，你不必担心两者之间的互联网连接会出什么问题。当连接恢复时，RabbitMQ 会负责重连至主 RabbitMQ 集群，并开始将所有在连接关闭时从网站发来的消息入队等候。稍后片刻，下游消费者应当能够跟上节奏，我们甚至都不用动一动手指。这听起来是不是像在变魔术？也许吧，但是其实背后并没有什么魔法。

在 RabbitMQ 中，应用了联合策略的交换器有特殊的处理进程。当在交换器上应用了联合策略时，该交换器会连接到策略中定义的所有上游节点，并创建一个工作队列用来接收消息。之后该交换器进程会注册成为工作队列的消费者并等待消息的到达。下游节点中交换器的绑定会自动应用到上游节点中的交换器和工作队列上，以便上游 RabbitMQ 节点将消息发送至下游消费者。当下游消费者接收消息时，它会像其他消息发送者那样，将消息发送至本地交换器上。这些消息携带了额外的消息头，将被路由到适当的目的地（见图 8.4）。

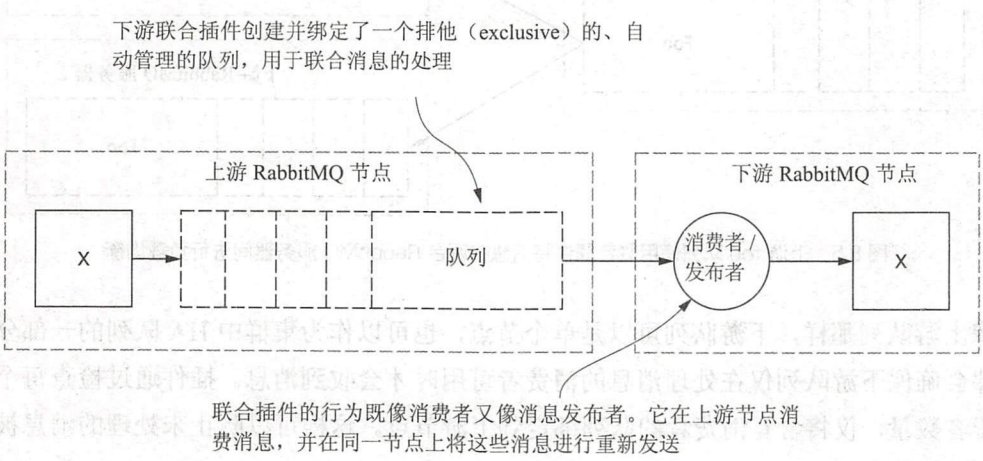


图 8.4　联合插件在上游 RabbitMQ 节点中创建了一个工作队列

正如你所看到的那样，联合交换器提供一种简单、可靠、健壮的方式来扩展 RabbitMQ 的基础设施，实现了 RabbitMQ 集群无法实现的跨网络延迟。此外，联合插件还能桥接逻辑上隔离的 RabbitMQ 集群，例如同一数据中心上运行着两个不同 RabbitMQ 版本的集群。

联合插件是一个强大的工具，为消息通信架构组成巨大的网络。针对更具体的需求，联合队列提供了更加细化的跨 RabbitMQ 集群的消息分发方式。这一方式允许轮询多个不同的下游节点和 RabbitMQ 消费者。

8.1.2 联合队列

联合插件的新增功能：基于队列的联合（*queue-based federation*），为扩展队列能力提供了方法。它对于解决消息通信负载特别有用。某些队列可能会有大量的消息发布活动，而消息消费的速度相对慢，或者受到了限流的影响。当采用联合队列时，消息发布者使用上游节点（集群），将消息发往下游节点的所有同名队列中（见图 8.5）。

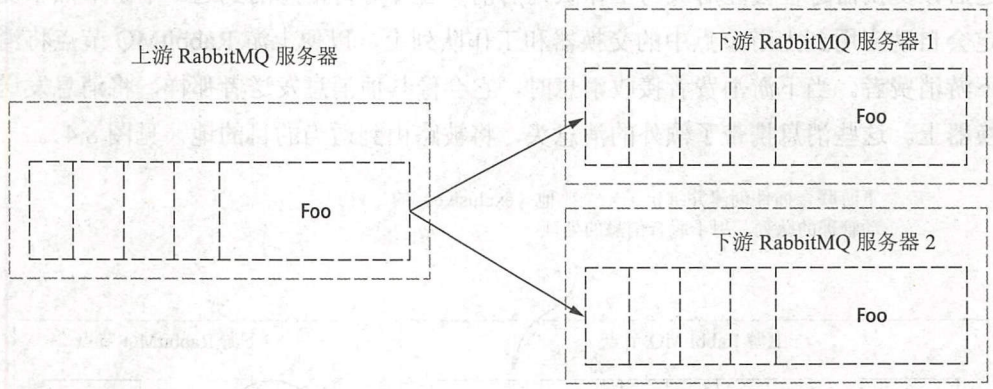


图 8.5　上游 foo 队列使用联合插件将消息在两台 RabbitMQ 服务器间进行负载均衡

像上游队列那样，下游队列可以是单个节点，也可以作为集群中 HA 队列的一部分。联合插件会确保下游队列仅在处理消息的消费者可用时才会收到消息。插件通过检查每个队列的消费者数量，仅将含有消费者的队列绑定到上游节点。这样可以防止未处理的消息被无消费者的队列接收。

在阅读本章后续介绍的配置相关的示例，你会发现：联合队列的配置和联合交换器的配置之间几乎没有差别。事实上，默认的联合配置同时适用于上述两者。

8.2　创建 RabbitMQ 虚拟机

本章的剩余部分，我们将使用免费的 Amazon EC2 服务器实例来搭建多台 RabbitMQ 服务器。我们将不用集群的方式，而是利用联合插件来透明地分发消息。即便你使用自己的云供应商或者之前就已存在的网络和服务器，这当中的理念是一样的。如果你不打算选择使用本章示例程序中使用的 AWS 的话，请在创建自己的服务器的同时尽可能地满足环境配置的要求。不论哪种场景，你应当搭建两台 RabbitMQ 服务器用于示例程序。

要在 Amazon EC2 上搭建 VM，首先需要创建服务器实例，安装并配置 RabbitMQ。然后创建该实例的镜像，以便在实验时需要创建更多的服务器副本。如果没有自己的虚拟服务器的话，那需要一个 AWS 账户。如果还没账户的话，可以免费在 http://aws.amazon.com 上创建一个。

8.2.1　创建首个实例

首先登录 AWS 控制台，单击 Create Instance。你将看到用于创建 VM 的镜像模板列表。在列表中选择 Ubuntu Server（见图 8.6）。

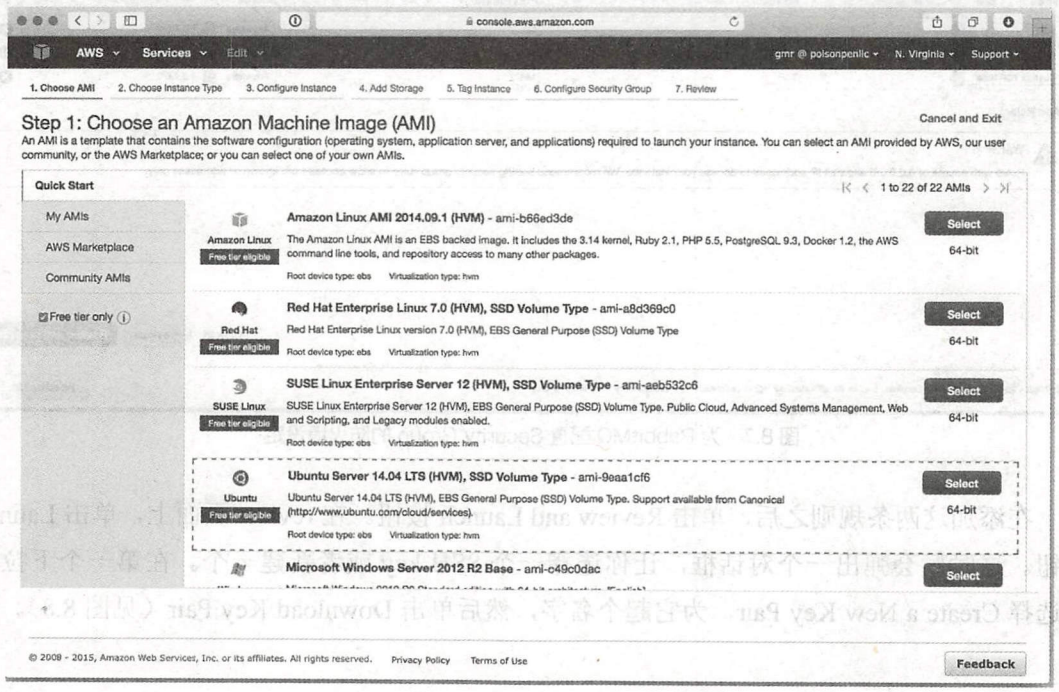


图 8.6　选择一个 AMI 来启动 Amazon EC2 实例

下一步是选择实例类型。选择通用的 t2.micro 实例。这一项应该被标记为 Free Tier Eligible。

当选择了这一类型的实例之后，你将会看到该实例的配置信息。你可以采用默认配置并单击 Next: Add Storage 按钮。同样的，这里仍然采用默认配置，并单击 Next: Tag Instance 按钮。该页面上无需做任何更改。单击 Next: Configure Security Group，你将看到安全组配置页面。你需要修改这里的配置，以和 RabbitMQ 进行通信。由于仅是实验目的，可以开放 5672 和 15672 端口而不用设置任何资源限制。单击 Add Rule 按钮，应用这条新的防火墙规则；同时，将每个端口的 Source 设置为 Anywhere，如图 8.7 所示。

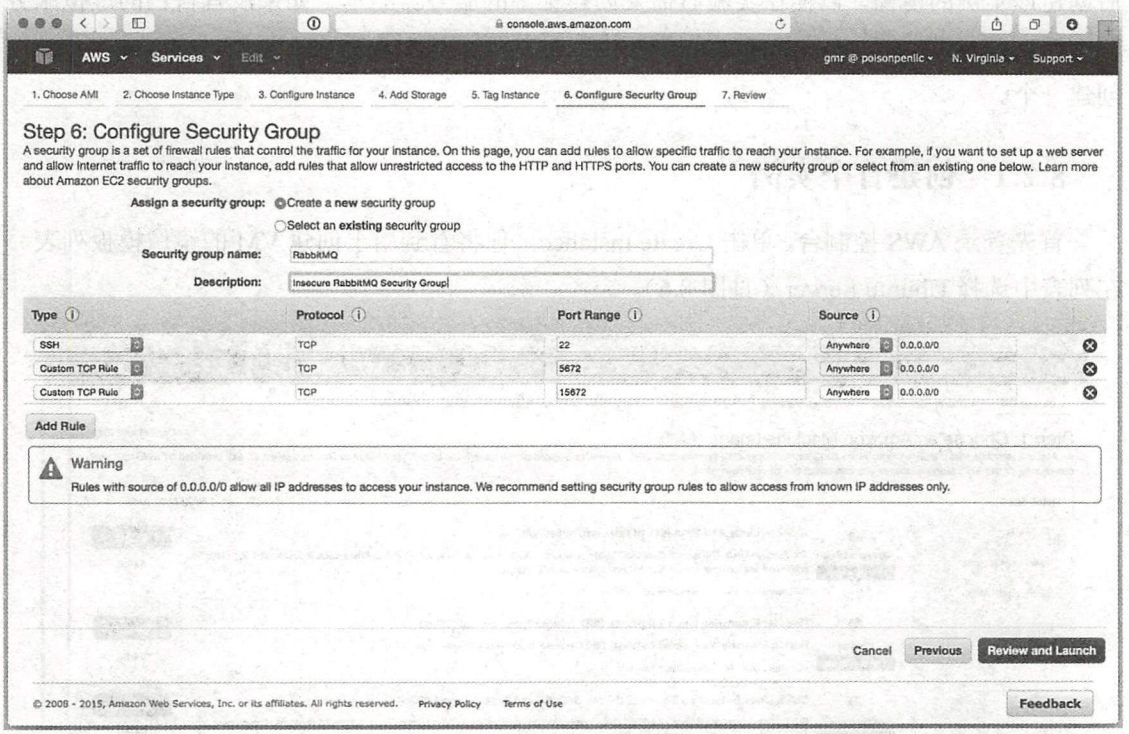


图 8.7　为 RabbitMQ 配置 Security Group 的防火墙设定

在添加这两条规则之后，单击 Review and Launch 按钮。在 review 页面上，单击 Launch 按钮。这时候会弹出一个对话框，让你选择一个 SSH key 或者新建一个。在第一个下拉框里选择 Create a New Key Pair，为它起个名字，然后单击 Download Key Pair（见图 8.8）。将

密钥对存放在本机可访问的位置，后面要用于 SSH 到 EC2 实例上。

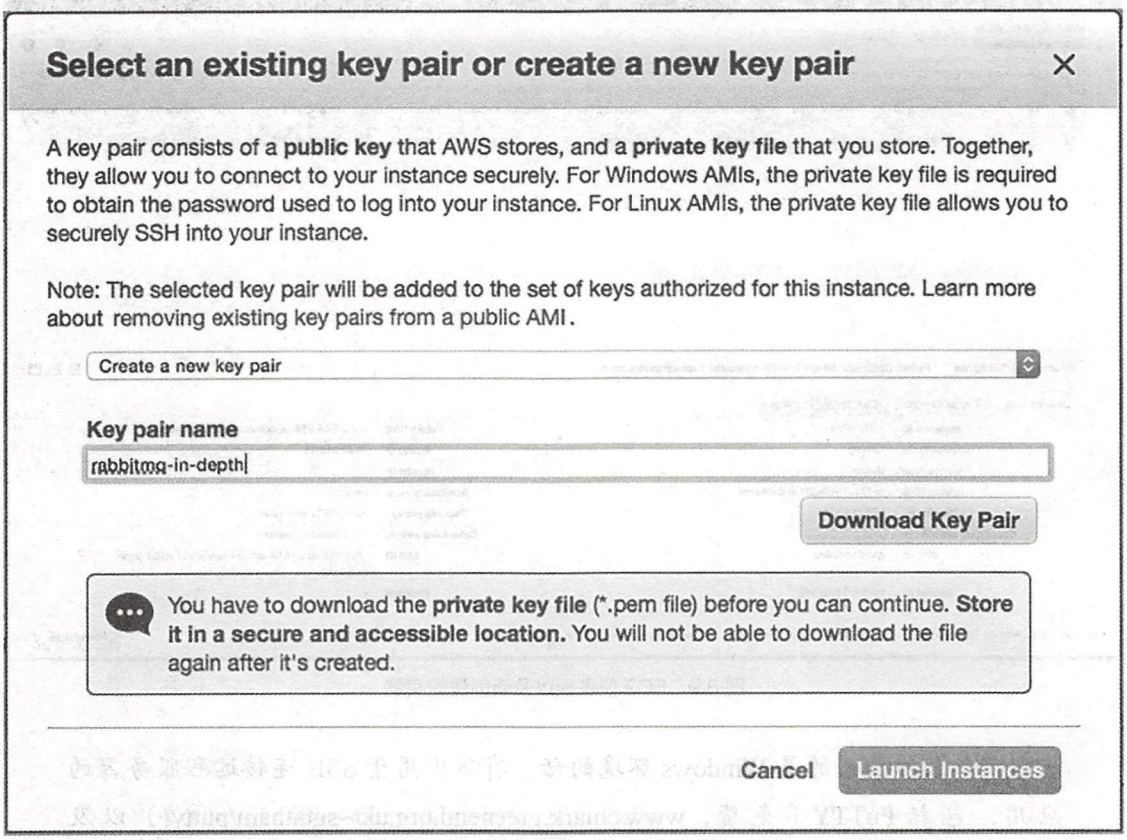


图 8.8　新创建密钥对，用来访问 VM

在下载了密钥对之后，单击 Launch Instance 按钮。AWS 将会创建并启动新建的 VM 实例。重新切换到 EC2 的仪表板上，你将看到这个新实例正在启动或者正在运行（见图 8.9）。

在 EC2 实例启动完成后，它将拥有一份公网 IP 地址和 DNS。记录下这个 IP 地址，需要用它来连上 VM 配置 RabbitMQ。

连接 EC2 实例

有了 EC 实例的 IP 地址和 SSH 密钥对的地址之后，现在就可以 SSH 到 VM 上，开始配置 RabbitMQ 了。以 ubuntu 用户进行连接，你需要指定 SSH 密钥对的路径。以下命令指向了本机 home 目录下的 Download 文件夹的命令如下。

```
ssh -i ~/Downloads/rabbitmq-in-depth.pem.txt ubuntu@[Public IP]
```

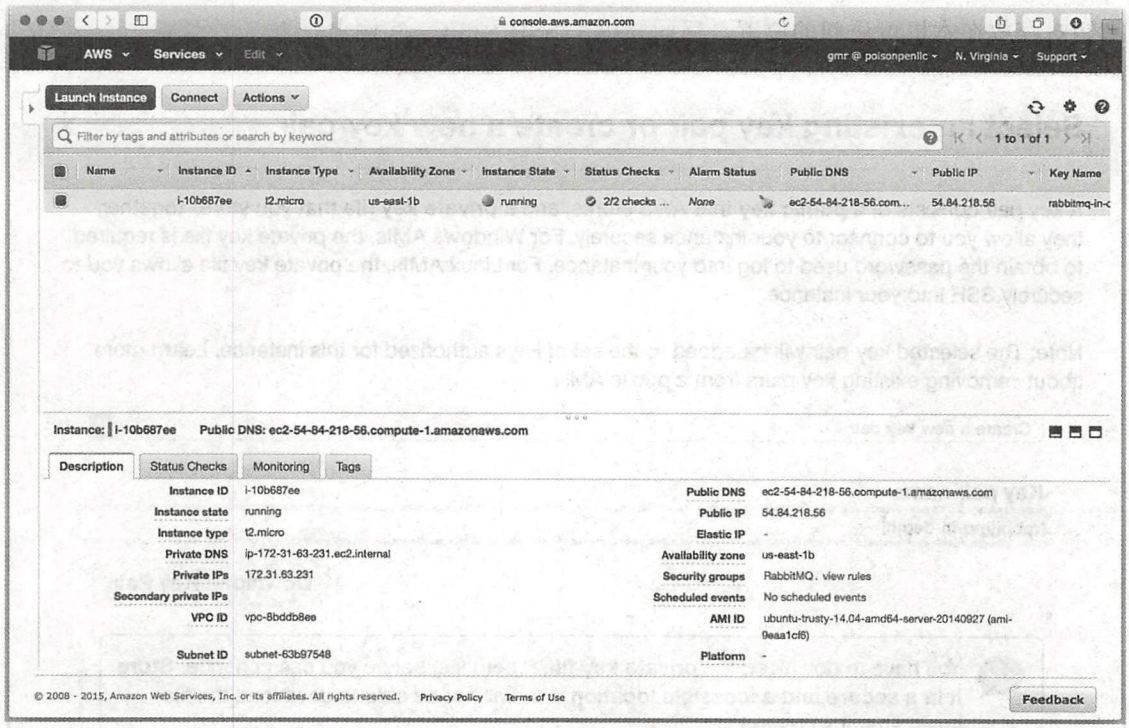


图 8.9　EC2 仪表板以及新创建的实例

注意　如果你使用的是 Windows 环境的话，有不少用于 SSH 连接远程服务器的应用，包括 PuTTY（免费，www.chiark.greenend.org.uk/~sgtatham/putty/）以及 SecureCRT（商业应用，www.vandyke.com/products/securecrt/）。

一旦连接成功之后，请以 ubuntu 用户身份登录。你可以看到类似下面的 MOTD 标语：

```
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-36-generic x86_64)
 * Documentation: https://help.ubuntu.com/
   System information as of Sun Jan 4 23:36:53 UTC 2015
   System load: 0.0       Memory usage: 5% Processes:   82
   Usage of /: 9.7% of 7.74GB Swap usage: 0% Users logged in: 0
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by applicable
law.
ubuntu@ip-172-31-63-231:~$
```

由于有不少命令需要你以 root 用户的身份发出，请切换至 root 用户，这样就不用总是敲 sudo 命令了：

```
sudo su -
```

作为 root 用户，你现在可以在这台 EC2 实例上安装 Erlang 运行环境和 RabbitMQ 了。

安装 Erlang 和 RabbitMQ

你可以采用官方 RabbitMQ 和 Erlang Solutions 仓库来进行安装。虽然主 Ubuntu 包仓库已经支持了 RabbitMQ 和 Erlang，但它们经常会严重滞后，因此我还是建议你获取它们的最新版本。为了使用外部仓库，你需要为外部仓库添加包签名密钥和配置。

首先，添加 RabbitMQ 公共密钥，以便 Ubuntu 能够验证安装包的文件签名：

```
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
    --recv 6B73A36E6026DFCA
```

可以在 apt-key 应用输出中看到引入了“RabbitMQ Release Signing Key <info@rabbitmq.com>”。

在将密钥导入到受信包密钥数据库后，现在就能为 Ubuntu 添加官方的 RabbitMQ 包仓库了。下列命令将在合适的位置添加一份新文件，用于添加 RabbitMQ 仓库：

```
echo "deb http://www.rabbitmq.com/debian/ testing main" \
     > /etc/apt/sources.list.d/rabbitmq.list
```

现在我们配置好了 RabbitMQ 仓库，接下来需要将 Erlang Solutions 的密钥添加到受信密钥数据库中：

```
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv \
     D208507CA14F4FCA
```

在 apt-key 命令完成后，你将看到“Erlang Solutions Ltd. <packages@erlang-solutions.com>”这份密钥被导入进来。

现在添加 Erlang Solutions Ltd. 仓库配置：

```
echo "deb http://packages.erlang-solutions.com/debian precise contrib" \
     > /etc/apt/sources.list.d/erlang-solutions.list
```

一切就绪，下面的命令将同步本地包数据库，这样一来就可以安装 RabbitMQ 了：

```
apt-get update
```

现在开始动手安装。rabbitmq-server 包将自动解析 Erlang 依赖并安装合适的包。

```
apt-get install -y rabbitmq-server
```

在该安装命令结束之后，RabbitMQ 就应该已经安装好并启动了。且慢，还需要运行一些命令来启用合适的插件，并开放 AMQP 连接端口和管理界面。

配置 RabbitMQ

虽然 RabbitMQ 包含了服务器管理和运行联合插件所需的全部功能，但默认情况下没有启用。配置 RabbitMQ 实例的第一步是启用随 RabbitMQ 一同发布的插件。启用之后就能配置使用联合的能力了。要实现上述目标，需要运行 rabbitmq-plugins 命令：

```
rabbitmq-plugins enable rabbitmq_management rabbitmq_federation \
    rabbitmq_federation_management
```

RabbitMQ 3.4.0 将自动加载插件而无须重启服务器。需要启用默认的 guest 用户，以便从 localhost 之外的其他 IP 登录。明确了需求之后，就可以在 /etc/rabbitmq/rabbitmq.config 目录下创建 RabbitMQ 配置文件，文件内容如下：

```
[{rabbit, [{loopback_users, []}]}].
```

需要重启 RabbitMQ，以使 loopback_users 设置生效：

```
service rabbitmq-server restart
```

为了防止同时使用两个 VM 而导致的混淆，可以使用 rabbitmqctl 命令修改集群名称。在管理界面的右上角显示了集群名称。使用如下命令来设置集群名称：

```
rabbitmqctl set_cluster_name cluster-a
```

现在来测试安装和配置是否正确。在浏览器输入 EC2 服务器的 IP 地址和端口号 15672 打开管理界面，形如 http://[PublicIP]:15672。在以 guest 用户（密码 guest）登录之后，你将看到 Overview 界面（见图 8.10）。

第一个示例创建好了，可以利用 Amazon EC2 仪表板从刚创建的那个运行中的实例创建镜像，再启动另一个 VM。新的镜像让我们很容易创建新的、预先配置好的、独立 RabbitMQ 服务器，用来测试 RabbitMQ 的联合能力。

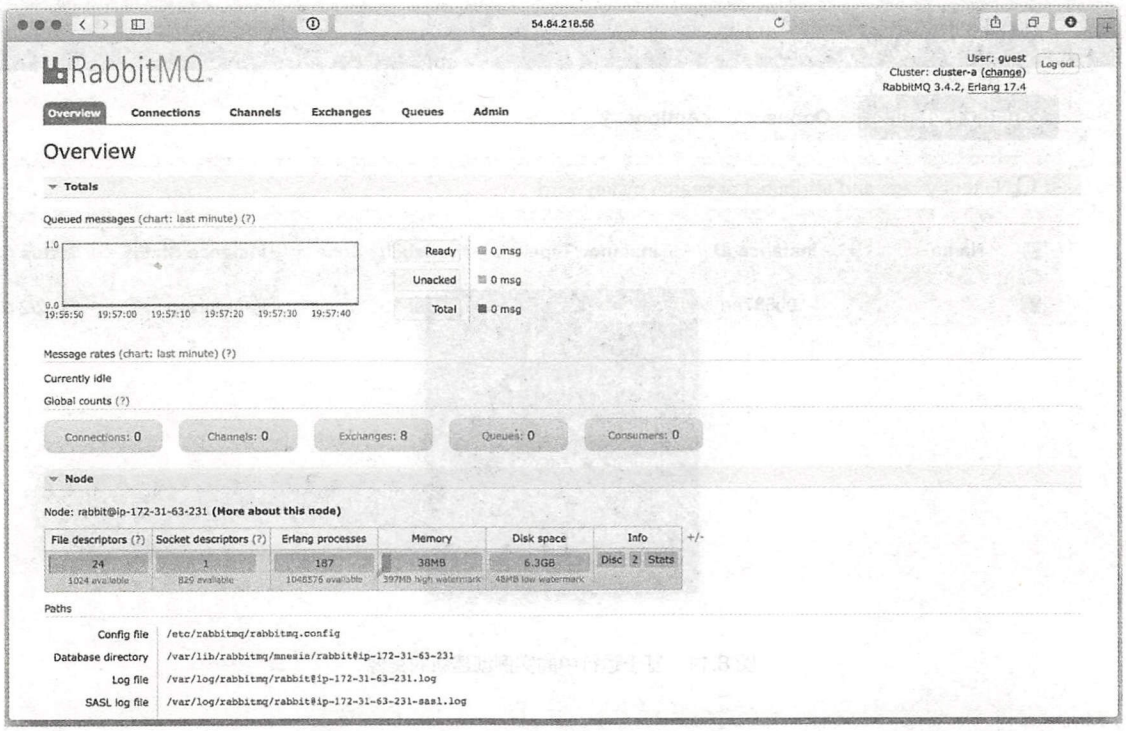


图 8.10　RabbitMQ 管理界面中的 Overview 视图

8.2.2　复制 EC2 实例

为了避免重复之前 RabbitMQ 独立实例的创建步骤，我们把这个工作交给 Amazon 来完成。要实现这一点，只要告诉 EC2 从运行中的 VM 实例来创建新的镜像（AMI）即可。

在浏览器窗口中访问 EC2 Instances 仪表板，单击运行中的实例。菜单会感知上下文，弹出该实例上所允许的操作。我们在菜单单击进入 Image > Create Image（见图 8.11）。

单击 Create Image 之后会弹出一个对话框，可以在该对话框中设置镜像的参数。给镜像起个名字，其他的选项不用动。在单击 Create Image 按钮之后，现有的这台 VM 将会被关闭，该 VM 的磁盘镜像将被用来创建新的 AMI（见图 8.12）。

当系统为原始 VM 创建文件系统镜像后，它将自动重启。同时，创建 AMI 的任务将会被添加到 Amazon 的系统队列中。AMI 可用需要花费几分钟的时间。你可以通过单击边上导航栏中的 Images > AMIs 选项来检查状态（见图 8.13）。

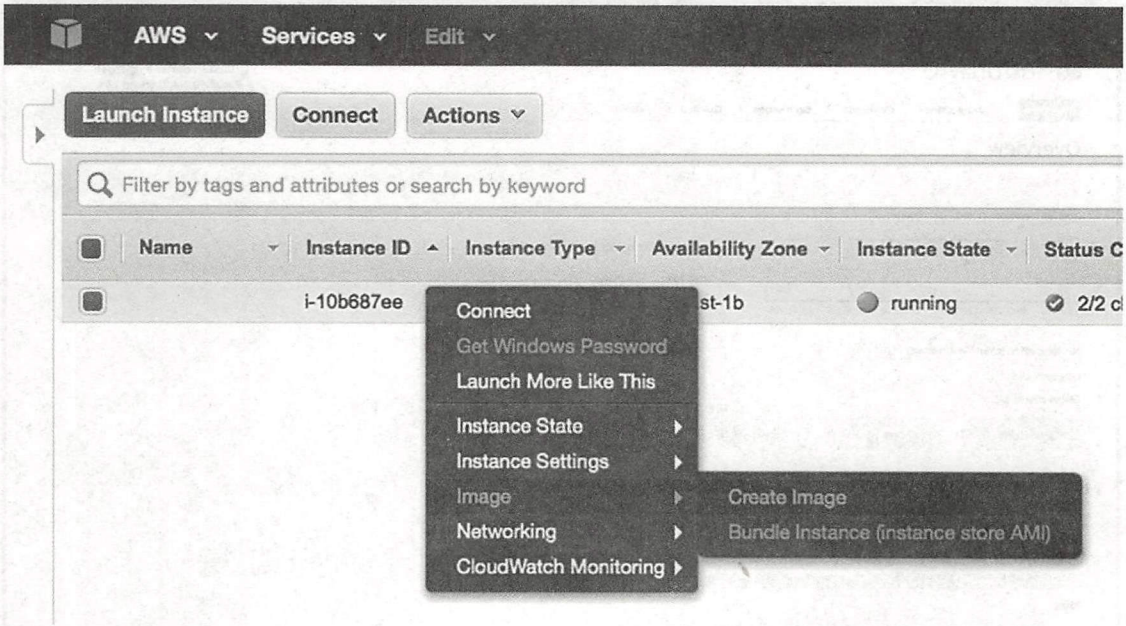


图 8.11　基于运行中的实例创建新的镜像

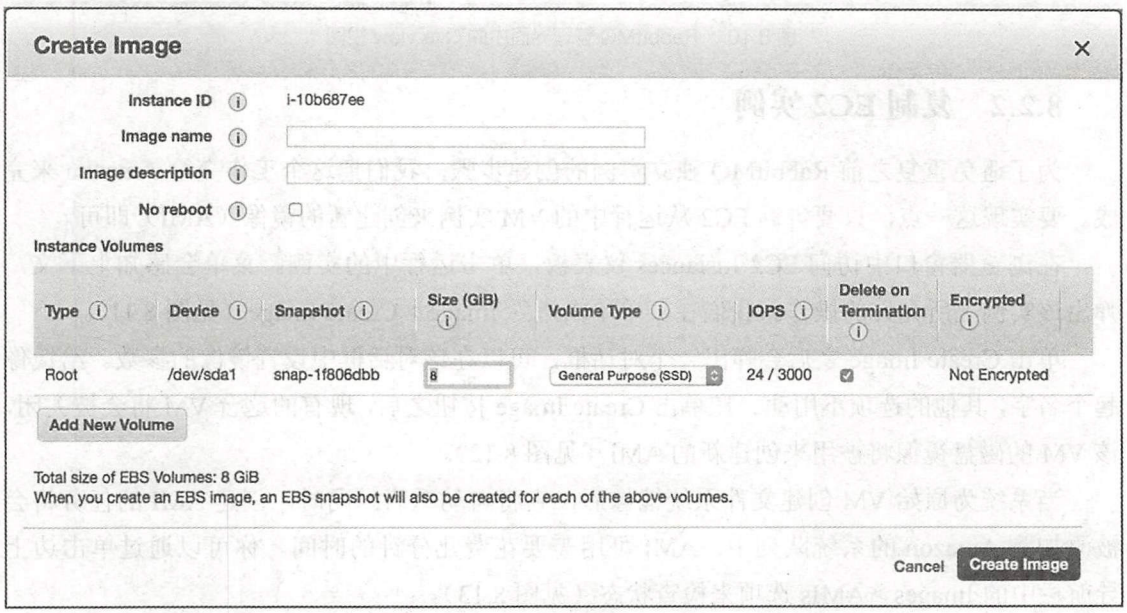


图 8.12　Create Image 对话框

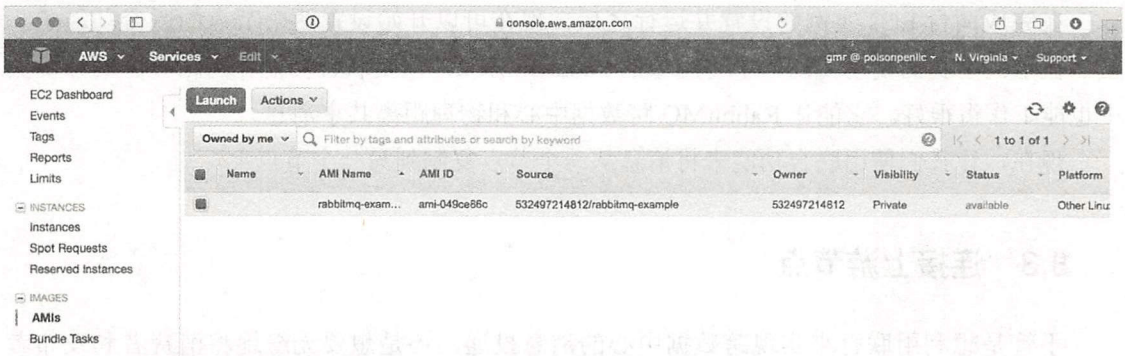


图 8.13　EC2 仪表板的 AMI 模块

当 AMI 可用之后，在 AMI 仪表板上选中它，然后单击顶部的 Launch 按钮，启动第二台 VM。请重复执行刚才在第一台 VM 上的所有步骤。不过不用再新建安全策略和 SSH 密钥对了，我们使用第一台 VM 的即可。

在 VM 创建完成之后，让我们回到 EC2 实例仪表板。等待实例启动成功后，记录下它的公网 IP。在 VM 开始运行之后，可以在浏览器里输入形如 http:// [Public IP]:15672 的 URL 地址来访问管理界面。登录到管理界面，单击右上角的 Change 链接将集群名称修改为“cluster-b”（见图 8.14）。修改集群名称有助于区分当前使用的是哪台服务器的管理界面。

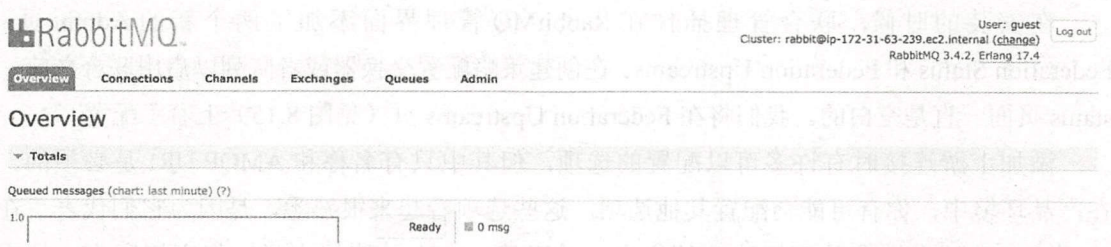


图 8.14　管理界面的右上角展示了集群的名称

在完成两台 EC2 实例的设置并运行之后，现在可以开始设置两个节点的联合功能了。虽然在示例中我们使用了同一个可用域的 Amazon EC2，但是联合功能在网络分区的环境当中也能工作得很好。它能让 RabbitMQ 跨数据中心和物理距离共享消息。

现在，让我们使用联合功能来将消息从一个节点复制到另一个节点上。

8.3 连接上游节点

不管是想利用联合来实现跨数据中心的消息投递，还是想要无缝地将消费者和发布者迁往新的 RabbitMQ 集群，首先要解决的问题是上游配置。虽然上游节点负责向下游节点投递消息，但是配置要在下游节点上完成。

联合配置由两部分组成：上游配置和联合策略。首先，下游节点需要配置用来发起 AMQP 连接到上游节点的信息。然后是策略配置，将上游连接和配置选项应用到下游交换器或者队列上。单台 RabbitMQ 服务器可以有多个联合上游和多个联合策略。

为了让下游节点 cluster-b 接收来自上游节点 `cluster-a` 的消息，首先必须在 RabbitMQ 管理界面里定义上游节点。

8.3.1 定义联合中的上游节点

在安装的时候，联合管理插件在 RabbitMQ 管理界面添加了两个新的 Admin 页：Federation Status 和 Federation Upstreams。在创建策略配置交换器或者队列以启用联合之前，status 页面一直是空白的。我们将在 Federation Upstreams 页（见图 8.15）上着手配置。

添加上游连接时有许多可以配置的选项，但其中只有名称和 AMQP URI 是必须的。在产品环境中，你有可能会配置其他选项。这些选项看起来很熟悉，是因为它们代表了在队列定义和消息消费的可用选项的组合。对于第一个上游连接来说，将它们留空，使用 RabbitMQ 默认设置即可。

上游节点的连接信息的定义需要输入远程服务器的 AMQP URI。AMQP URI 规范允许连接的灵活配置，包括可以微调心跳间隔、最大帧大小、连接端口、用户名和密码等。完整的 AMQP URI 语法，包括可用的查询参数，可以在 RabbitMQ 网站 www.rabbitmq.com/uri-spec.html 上找到。由于我们使用的测试环境很简单，除了 URL 中的主机名之外的其他配置都可以使用默认值。

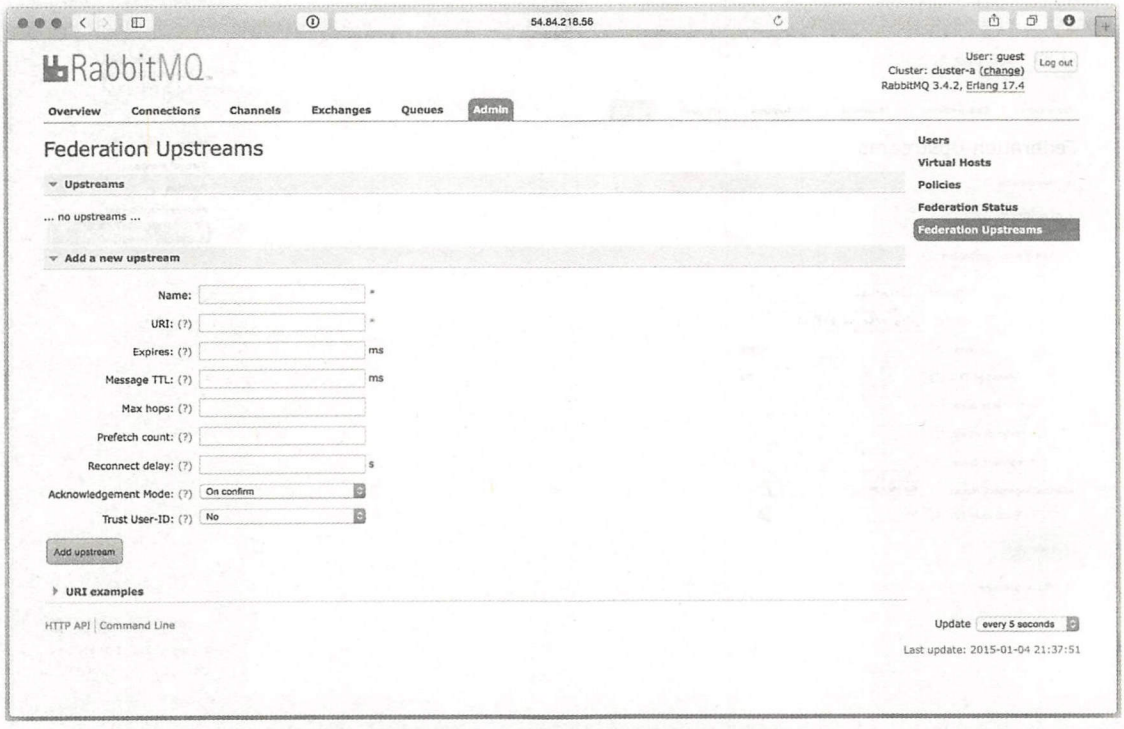


图 8.15　管理界面中 Admin 区域中的 Federation Upstreams

在测试环境中，`cluster-b` 节点将扮演下游节点的角色并连接到 `cluster-a`。在浏览器打开管理界面并导航到 Admin 区域的 Federation Upstreams 页。展开 Add a New Upstream 区域，输入 `cluster-a` 作为上游节点的名称。URI 部分请输入 amqp://[PublicIP]，将 [PublicIP] 替换为本章设置的第一个节点的 IP 地址（见图 8.16）。

此处输入的信息定义了通往另一个 RabbitMQ 节点的单条连接，在创建指向该上游节点的策略后生效。当策略中使用了上游节点时，联合插件会连接到上游节点。如果由于路由错误或者其他网络问题导致连接断开，默认的行为会每隔一秒钟进行重连。你可以在定义该上游节点时调整 Reconnect Delay 字段来更改这一行为。如果想要在上游节点创建完成之后来改变这一行为的话，那就必须先删除，再重新创建。

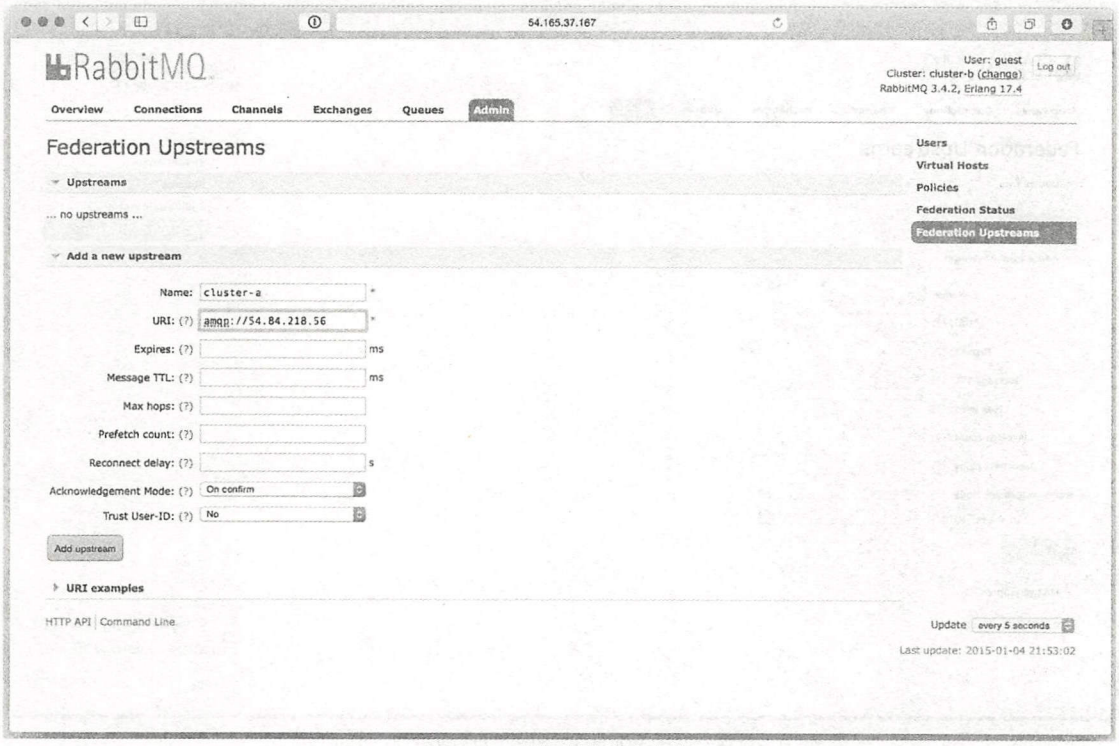


图 8.16　添加新的联合上游

在输入名称和 URI 之后，单击 Add Upstream 保存上游配置。添加上游完成之后就可以定义策略，测试基于交换器的联合。

注意　虽然本章的示例程序使用管理界面来创建上游节点，但也可以使用 HTTP 管理 API 和 `rabbitmqctl` CLI 应用程序来实现。有关使用 rabbitmqctl 添加上游的示例，请访问联合插件文档 http://rabbitmq.com。

8.3.2　定义策略

联合的配置是由 RabbitMQ 的策略系统来管理的。策略提供了一种灵活的方式，可以动态地配置规则，以改变联合插件的行为。在创建策略时需要首先明确策略的名称和模式。这里的模式可以是直接字符串匹配，也可以提供一个正则表达式（regex）模式，用来匹配 RabbitMQ 对象。模式可用来分别与交换器、队列进行，也可以和两者同时作比较。策略还可以指定优先级，用来决定在匹配多个策略时应当在队列或者交换器上应用哪个策略。当队

列或交换器匹配了多个策略时，拥有最高优先级的策略将胜出。最后，每个策略都有一张定义表，可以在里面定义任意的键值对。

对于第一个示例来说，你将创建一个名为 federation-test 的策略。它将根据字符串相等的规则检测名为 test 交换器（见图 8.17）。在定义表中添加键 federation-upstream 和值 cluster-a。联合插件将通过该数据联合上游 cluster-a 的交换器。输入完信息后，单击 Add Policy 按钮添加到系统中。

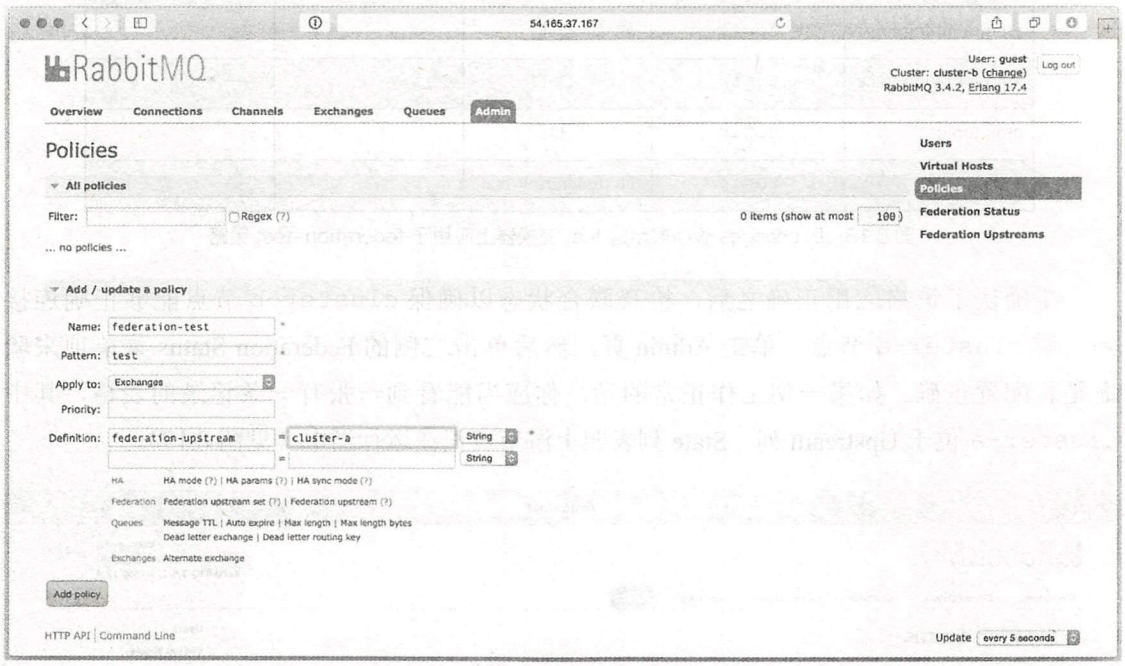


图 8.17　新建策略，用来联合上游节点 cluster-a

在添加完策略之后，你需要在这两个节点中添加 test 交换器。可以使用每个节点上管理界面中的 Exchanges 页来添加交换器。为了防止 cluster-b 节点尝试联合到 cluster-a 节点上一个不存在的交换器上，首先应在 cluster-a 节点上声明交换器。你可以选用任何内建的交换器类型。但出于灵活性的考虑，我推荐使用 topic 类型的交换器。不管选择了何种类型，应当确保在 cluster-a 节点和 cluster-b 节点上创建的 test 交换器拥有相同的交换器类型。

一旦添加完交换器后，在 cluster-b 节点上管理界面 Exchanges 页的 Feature 栏中，test 交换器将会拥有一个匹配了联合策略的标签（见图 8.18）。该标签表明已经成功地将策略匹配到了交换器上。

名　称	类　型	特　征	消息接收速率	消息发送速率
(AMQP defaut)	direct	D		
amq.direct	direct	D		
amq.fanout	fanout	D		
amq.hexders	headers	D		
amq.match	headers	D		
amq.rabbitmg.log	topic	D I		
amq.rabbitmg.trace	topic	D I		
amq.topic	topic	D		
test	topic	D federation-teat		

图 8.18　Exchanges 表格展示的 test 交换器上应用了 federation-test 策略

在确认了策略应用正确之后，检查联合状态以确保 cluster-b 节点能够正确连接到上游 cluster-a 节点。单击 Admin 页，然后单击右侧的 Federation Status 菜单项来验证是否配置正确。如果一切工作正常的话，你应当能看到一张有一条记录的表格，其中 cluster-a 位于 Upstream 列。State 列表明上游节点正在运行当中（见图 8.19）。

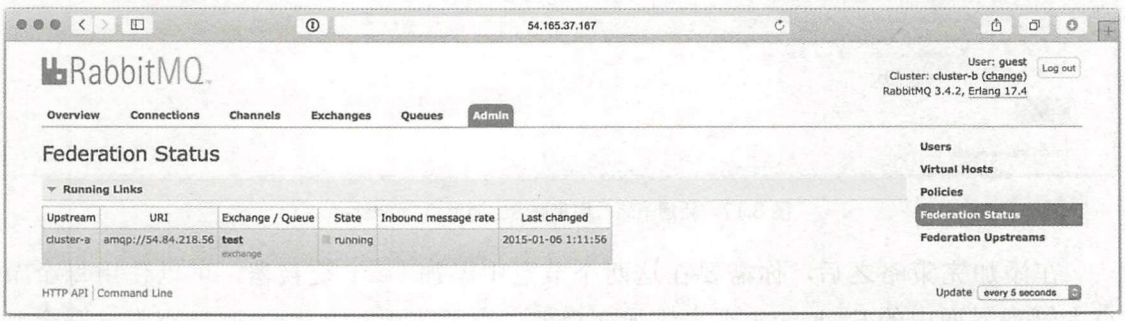


图 8.19　Federation Status 页表明上游 cluster-a 节点正为 test 交换器运行

在确认 RabbitMQ 已经一切就位后，你可以通过向 cluster-a 发送一条消息，该消息会在 cluster-b 上入队。为了实现上述目的，在 cluster-b 上创建一个测试队列，并将其绑定到 test 交换器，绑定键为 demo。这样就在本地 cluster-b 节点和 cluster-a 节点的联合消息 test 交换器上都建立了绑定。

切回 cluster-a 的管理界面，在 Exchange 页上选择 test 交换器。在 test 交换器页面上，展开 Publish Message 部分。输入路由键 demo，在 Payload 字段填上你喜欢的内

容。在单击 Publish Message 按钮之后，这一消息将被发往 cluster-a 和 cluster-b 上的 test 交换器，并且该消息应当处于 cluster-b 节点上的 test 队列中。

回到 cluster-b 管理界面，切换到 Queues 页，选择你的 test 队列。展开 Get Messages 部分并单击 Get Message(s) 按钮。你应当能够看到之前发往 cluster-a 的消息（见图 8.20）

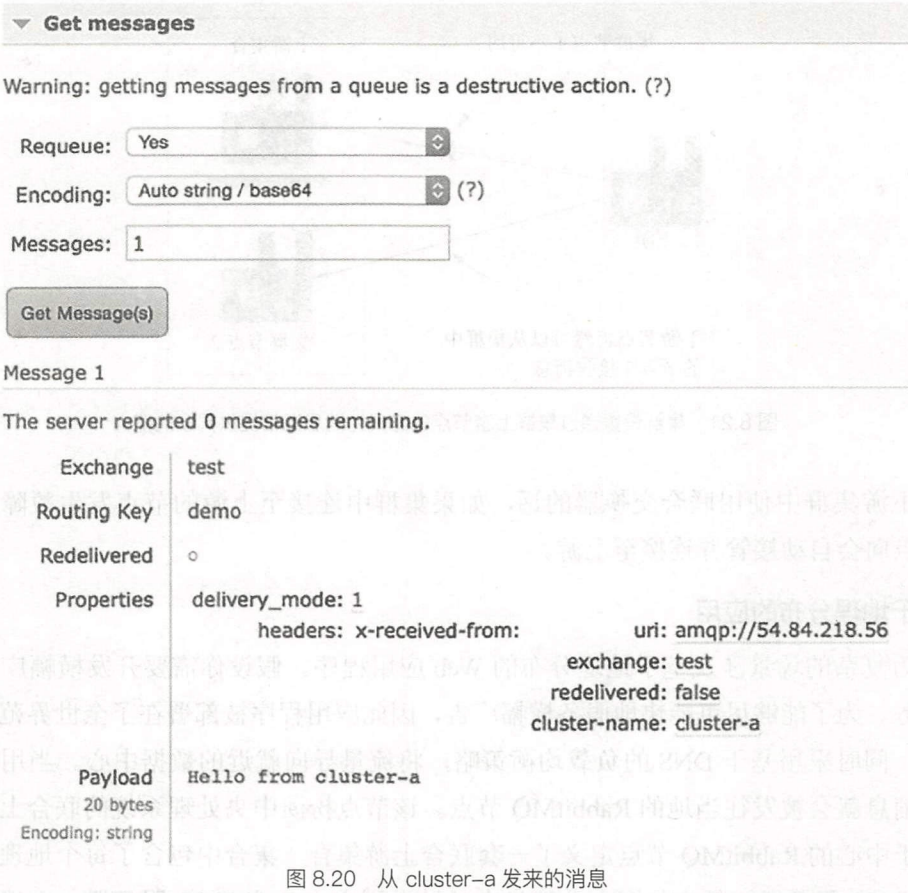


图 8.20　从 cluster-a 发来的消息

为了识别出那些通过联合方式发送过来的消息，联合插件会在消息属性的 headers 表中添加一个 x-received-from 字段。该属性的值是一张键值对表，包括上游的 uri、exchange、cluster-name，以及一个用于表示消息是否是 redelivered 的标记。

8.3.3　利用上游集合

除了定义单个上游节点外，联合插件还提供了使用策略将多个节点分组的能力。这一分组功为联合拓扑的定义提供了相当多的功能。

提供冗余能力

举例来说，假设上游节点是集群中的一部分。你可以创建一个上游集合，定义每一个上游集群节点，允许下游节点连接到该集群中的每个节点，确保如果当中有一个节点发生故障的话，发往上游集群的消息在发往下游节点时不会丢失（见图 8.21）。

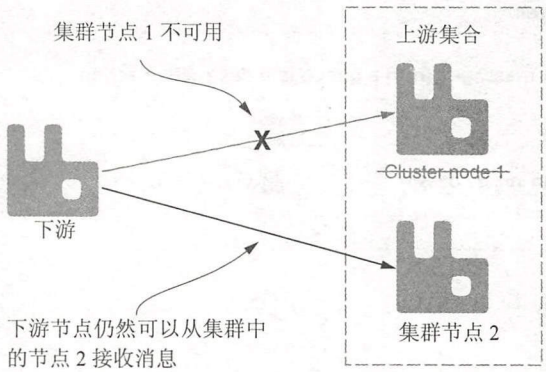


图 8.21　集群设置通过集群上游节点的方式为消息通信提供冗余的能力

在下游集群中使用联合交换器的话，如果集群中连接至上游的节点发生故障，那么另一个节点向会自动接管并连接至上游。

基于地理分布的应用

更为复杂的场景涉及基于地理分布的 Web 应用程序。假设你需要开发横幅广告浏览记录的服务。为了能够尽可能快地服务横幅广告，因此应用程序被部署在了全世界范围内的每个角落，同时采用基于 DNS 的负载均衡策略，将流量导向就近的数据中心。当用户浏览广告时，消息就会被发往当地的 RabbitMQ 节点。该节点扮演中央处理系统的联合上游节点角色。位于中心的 RabbitMQ 节点定义了一套联合上游集合。集合中包含了每个地理分布位置的 RabbitMQ 服务器。每个位置上的消息会被转发到中央 RabbitMQ 服务器，由消费者应用程序进行处理（见图 8.22）。

正因为联合插件有着类似客户端的行为，容许连接故障，任何一个地理分布的节点离线时，对于来自系统中剩余部分流量的处理不会受到影响。要是区域性路由出现了问题，所有来自断线上游节点的入队消息会在下游节点重连时被发送出去。

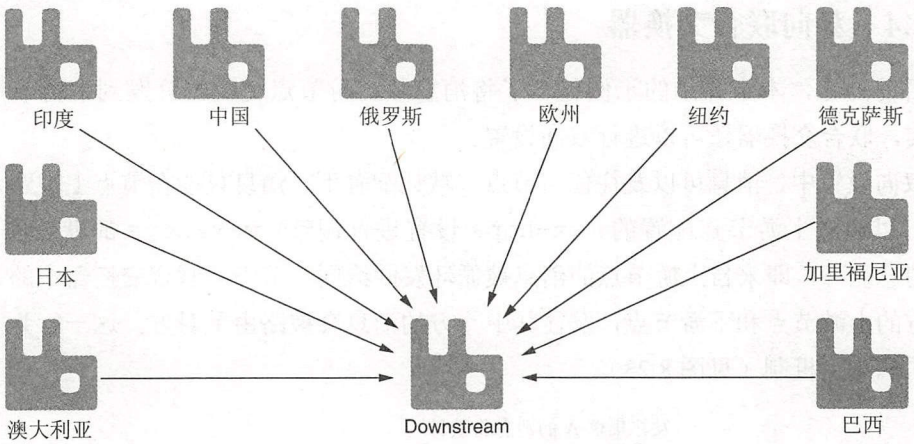


图 8.22　地理分布的上游节点集合，将消息发往下游节点

创建上游节点集合

要创建上游集合，首先需要通过管理界面或者 rabbitmqctl CLI 应用分别定义每一个上游节点。由于联合管理界面中没有创建上游集合的界面，你必须使用 rabbitmqctl 命令行工具，和往常一样在本地运行 rabbitmqctl 来配置管理 RabbitMQ 节点，并且需要有访问 RabbitMQ 的 Erlang cookie 权限。这一点我们在 7.2.2 章节中曾讨论过。

在定义了上游节点后，需要创建一个 JSON 字符串，包含创建上游节点定义时用到的名称列表。举例来说，假设你创建了名为 a-rabbit1 和 a-rabbit2 的上游节点的话，那么 JSON 片段如下所示：

```
[{"upstream": " a-rabbit1"}, [{"upstream": " a-rabbit2"}]
```

然后，定义一个名为 cluster-a 的上游节点，运行 rabbitmqctl 命令 set_paramete。该命令可以用来定义名为 cluster-a 的 federation-upstream-set。

```
rabbitmqctl set_parameter federation-upstream-set cluster-a \
      '[{"upstream": " a-rabbit1"}, {"upstream: " a-rabbit2"}]'
```

在定义了上游集合后，你可以引用集合的名字 federation-upstream-set 来作为联合策略的键，来取代之前创建单个节点时使用的 federation-upstream 键。

值得注意的是，有个隐式定义的上游集合 all，它不需要任何配置就包括了所有已定义的联合上游节点。

8.3.4 双向联合交换器

到目前为止，本章介绍的示例包含了将消息从上游节点交换器分发到下游节点交换器上。其实，联合交换器还可以进行双向设置。

在双向设置中，消息可以发往任一节点。默认配置下，消息只能在节点上被路由一次。该设定可以通过上游节点配置的 max-hops 设置进行调整。max-hops 的默认值是 1，用来防止消息循环，即来自上游节点的消息被循环发回给同一节点。联合交换器中的每个节点互为对方的上游节点和下游节点，发往其中一方的消息会被路由至对方。这一行为类似于集群中的消息路由机制（见图 8.23）。

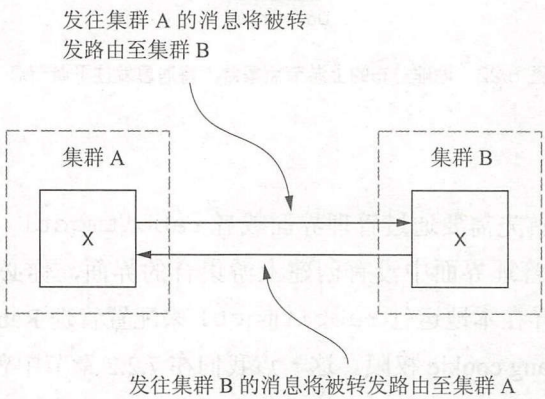


图 8.23 发往任一节点双向交换器的消息将被路由至对方节点上

这一类型的联合适合创建容错、多数据中心应用结构。比起将数据跨数据中心或者地理位置进行分片，这一类型的联合可以让每个地理位置都能接收到相同的消息并加以处理。

虽然这是一种提供高可用服务的强大方法，但是同样也带来了额外的复杂性。当我们尝试使用联合为跨越不同地理位置保持数据的一致性视图时，那些围绕多主数据库而产生的复杂性和关切点也随之浮现。一致性管理对于确保数据在不同地理位置之间的一致性尤为重要。幸运的是，联合交换器能够提供一种简单的方式来实现不同地理位置之间的一致性消息通信。值得注意的是这一行为并不仅限于两个节点的情况，它也适用于所有节点互相连接形成的图（见图 8.24）。在两节点的配置中，将上游节点的 max-hops 设置成 1 可以防止消息在图中被重新发送而形成循环。

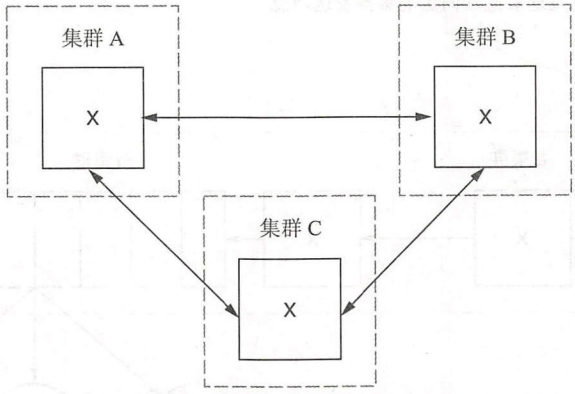


图 8.24　联合交换器可配置多于两个节点而形成的图

同样值得注意的是，就像其他图结构那样，越多的节点意味着越多的复杂性。就实现面向消息的体系结构的各个方面而言，应当在投入产品使用之前测试架构性能。幸好像 Amzon 这样的云服务供应商提供不同的可用区，可以方便地构建并测试复杂的 RabbitMQ 联合环境。

8.3.5　使用联合来升级集群

RabbitMQ 集群管理中一个让人颇为头疼的运营问题是如何在产品环境中处理升级问题。产品环境容不得一秒钟的停机时间。我们有不同的策略来应对这一场景。

当集群足够大时，你可以将一个节点的所有流量移走，把该节点从集群中移除，然后对它进行升级。之后再将另一个节点下线，从集群中移除，升级完毕之后，再将它添加到由之前移除的那个节点组成的新的集群当中。对集群中的每个节点重复这样的操作，将这些节点一个一个下线，直到所有节点都从集群中移除、升级并重新添加到新的集群当中为止。如果消息发送者和消费者能够平滑地处理重连的话，那么上述方法就能奏效。但是这种方法比较费力。如果你有资源来配置一套新版本的集群镜像的话，联合插件能将消息通信流量无缝地从一个集群切换到另一个。

当采用联合方式来升级 RabbitMQ 时，首先要做的是配置好新的集群，并创建相同的运行时配置，包括虚拟主机、用户、交换器和队列。在新集群配置完成之后，添加联合配置，包括上游节点和策略，匹配所有的交换器。之后，你就可以开始迁移消费者应用，将它们的连接从老的集群切换到新的集群中（见图 8.25）。

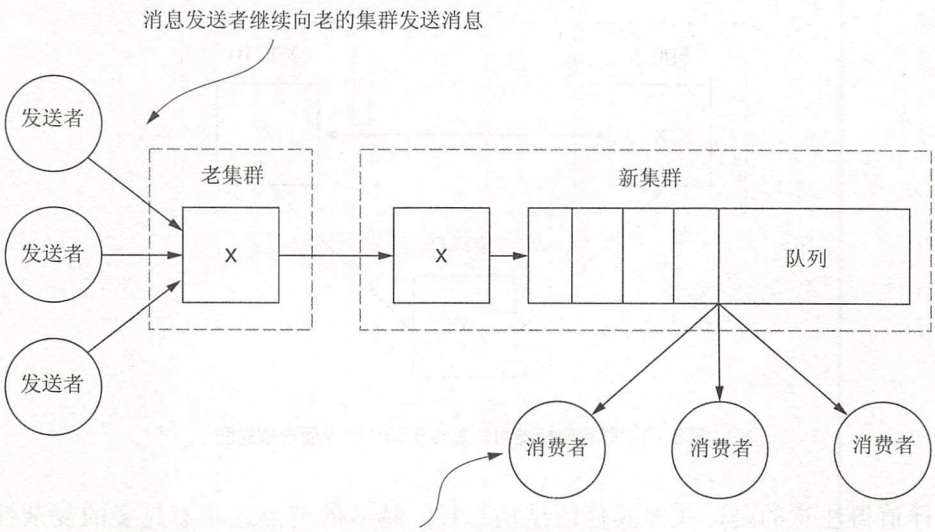


图 8.25　使用联合来升级 RabbitMQ 集群的第二阶段

当迁移消费者时，你应当在老集群上解绑队列，但不要删除。你可以在新集群上创建一个临时策略用来联合那个队列，将消息从老集群移动至新集群中。将这一过程尽可能自动化是明智之举，因为这样做可以最小化由于同时使用联合交换器和联合队列而导致重复消息被添加至新集群的队列中。

在将所有消费者迁移完毕并将队列从老集群中解绑之后，接下来就该迁移发送者了。当所有发送者迁移之后，这就意味着完全迁移至升级后的 RabbitMQ 集群了。当然，也许你想要将联合保留一阵子，以确保没有发送者还连接在老的集群上（这不应当发生）。这样做能确保应用程序正常运行，同时你可以使用老集群节点上的 RabbitMQ 日志来监控连接的建立与断开。虽然联合插件本不是为此类任务而设计的，但事实证明对于无缝 RabbitMQ 升级来说它是最佳选择。

8.4　小结

联合插件的灵活和强大，超越你的想象。不管是透明地将流量从一个 RabbitMQ 集群迁往另一个，还是创建多数据中心的应用程序来跨节点共享所有消息，联合插件都是可靠、高效的解决方案。作为一种扩展工具，联合队列为极大地增加单个队列的容量提供了一种方式：定义一个上游节点和任意数量的下游节点。当联合插件和集群、HA 队列结合起来之后，不管是集群间的网络分区容错性，还是防止上游集合或者下游节点故障的容错性，都将不在话下。

故障时有发生。在第 9 章中，你将学习多种策略以应对问题发生时的监控和告警。

第三篇

集成与定制

RabbitMQ 并未止于 AMQP 和交换器。它还可以集成其他有意思的功能。本书的第三篇将介绍 MQTT 和 STOMP 协议，使用 HTTP 协议进行无状态消息推送，以及如何将 RabbitMQ 与 PostgreSQL 和 InfluxDB 进行集成。

第 9 章　使用替代协议

本章概要

- 使用 MQTT 协议的优势与方法
- 基于 STOMP 应用如何与 RabbitMQ 通信
- 如何使用 Web STOMP 直接从 Web 浏览器发起通信
- 如何使用 statelessd 通过 HTTP 协议向 RabbitMQ 发送消息

AMQP 0-9-1 这一健壮的协议可以满足大多数应用程序与 RabbitMQ 的通信需求。而对于特殊使用场景，我们有更好的选择。举例来说，移动设备由于其高延迟、不可靠的网络通信会给 AMQP 带来诸多问题。相对而言，某些应用场景下，客户端应用程序不愿维护长连接，但是却想高速发送消息。这时，基于状态的 AMQP 就显得过于复杂了。此外，一些应用程序可能已经支持消息通信了，但却没有采用 AMQP 协议。针对以上这些场景，RabbitMQ 强大的应用和插件生态系统使得它能够成为你消息通信架构的核心。

本章将介绍可以替代标准 AMQP 0-9-1 协议之外一些协议。其中，MQTT 协议适用于移动端应用。STOMP 相对于 AMQP 来说更为简单。Web 版的 STOMP 协议被设计用于 Web 浏览器。statelessd 适用于高速消息发送。

9.1 MQTT 和 RabbitMQ

消息队列遥测传输（MQ Telemetry Transpor，即 MQTT）协议是一种轻量级的消息通信协议，在移动端应用中应用广泛。RabbitMQ 通过插件机制来支持它。来自 IBM 的 Andy Stanford-Clark 和来自 Eurotech 的 Arien Nipper 于 1999 年发明了这款基于发布 / 订阅模式的 MQTT 协议。MQTT 被设计用来在资源约束的设备以及低带宽的环境下使用，而不必牺牲消息通信的可靠性。虽然没有 AMQP 那样功能强大，但是随着移动端应用的爆发式增长，MQTT 近几年来也风光无限。

近几年来，从移动端应用到智能汽车以及家庭自动化，MQTT 在这些主流场景中的应用占据着科技新闻的头条。Facebook 采用 MQTT 实现了自家移动端应用的实时消息通信和通知。2013 年，福特汽车公司联手 IBM，采用 IBM 基于 MQTT 的 MessageSight 产品线，为福特 Evo 概念车实现了智能汽车技术。商业家庭自动化产品的势头略减，但大量开源且基于开放标准的家庭自动化系统仍采用 MQTT，例如 FunTechHouse 项目（ www.fun-tech.se/FunTechHouse/）。同样在 2013 年，就像 AMQP 1.0 那年前一样，MQTT 被 OASIS 采纳为开放标准。OASIS 是一家非营利机构，它致力于鼓励开放标准的开发与应用。此举为 MQTT 提供了开放、厂商中立的依靠，为其将来的发展铺平了道路。

那么你该在消息通信架构中考虑采用 MQTT 协议吗？在回答这个问题之前，我们首先要对它的优劣利弊做个分析：架构是否能从 MQTT 的最后遗愿功能（Last Will and Testament，LWT）中获益呢？（LWT 使得客户端能够在无意间断开连接时，发送一条指定的消息）。也许你会触及到 MQTT 的最大消息长度：256MB。即使 RabbitMQ 的 MQTT 插件能够在 MQTT 和 AMQP 协议之间进行转换，这一点对于应用程序来说是透明的。为了能够合理地评估 MQTT，与 AMQP 作对比，深入理解协议的通信机制大有裨益。

9.1.1 MQTT 协议

AMQP 和 MQTT 有不少共同之处。毕竟，大多数消息通信协议中的许多理念是相通的，例如支持连接协商，包括认证和消息发布。但本质上来讲，不同的协议在结构上千差万别。不同于 AMQP 协议级别的交换器和队列，MQTT 仅定义了消息的发布者和订阅者。当然，如果你使用 RabbitMQ 的话，这种限制影响不大。这是因为发往 RabbitMQ 的 MQTT 消息被

当作 AMQP 消息那样处理，同时，消息订阅者也被当作 AMQP 消费者来进行处理。

虽然 RabbitMQ 支持 MQTT 开箱机用，但通过 MQTT 和 AMQP 发布的消息中存在差异，这反映了协议的价值主张。作为轻量级协议，MQTT 对于受限硬件、不可靠的网络环境更为友好，而 AMQP 则设计得更为灵活但是需要更健壮可靠的网络环境。如果你未充分考虑到这些差异之处的话，那么应用程序在同一消息通信架构下使用这两种协议时会遇到互操作问题。在本节中，我们将剖析 MQTT 消息构造，以及它可能对消息体系结构和应用程序所造成的影响。

消息结构

MQTT 的底层结构是一个被称为命令消息的消息结构，就像 AMQP 的底层帧结构一样。命令消息作为底层数据机构封装了 MQTT 消息（见图 9.1）。

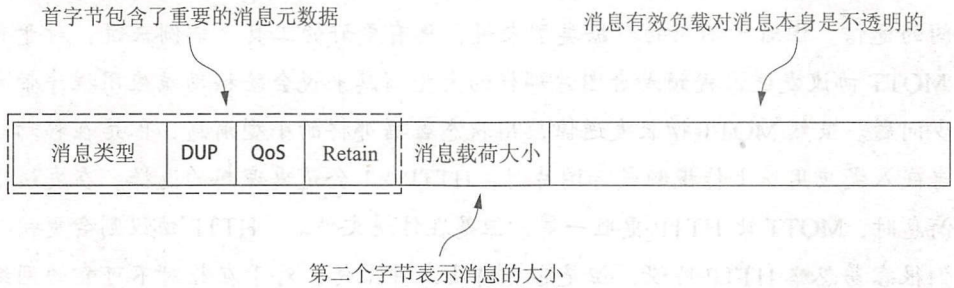


图 9.1　MQTT 命令消息的构造

MQTT 命令消息有一个固定两字节大小的头部用于描述消息。第一个头字节中包含的四个值分别是：

- 消息类型。共计四位，用于表示消息的动作，与 AMQP 方法帧类似。消息类型包括：CONNECT、PUBLISH 和 SUBSCRIBE。
- DUP 标志位。单独一位，用于表示该消息是否是重复投递，无须考虑客户端或者服务端是否正在重新投递该消息。
- QoS 标志位。共计两位，用于表示消息的服务质量。在 MQTT 中，QoS 明确了一条消息是否必须最多投递一次，还是至少投递一次，或者仅投递一次。
- Retain 标志位。单独一位，用于向服务器表示消息在发布给所有当前订阅者之后是否需要保留。MQTT 代理服务器仅会在 Retain 标志位被设置时保留最后一条消息，

这一机制可以让新订阅者总是能够收到最新一条消息。假设你在移动应用程序中使用了 MQTT 协议，如果应用程序和 RabbitMQ 服务器之间断开连接的话，通过 Retain 功能使得应用程序可以获取最近一条完整的消息，将对应用重连时重新同步状态有很大帮助。

MQTT 消息头中的第二个字节用于存放消息载荷的大小。MQTT 消息允许的最大载荷大小为 256MB。对比起来，AMQP 中最大的消息大小为 16EB，而 RabbitMQ 将消息大小限制为 2GB。在创建消息通信架构时，MQTT 的最大消息大小值得考虑，你可能会基于 MQTT 协议之上创建自己的协议，单条大于 256MB 的消息将被分割，并在订阅者端进行重新构建。

注意 根据马斯洛法则，如果手头有一把锤子，那么任何东西看起来都像是钉子。我们可以轻松地将 MQTT 或者 AMQP 这样的协议当作锤子一样用来解决应用间的通信。但对于不同的数据类型来说，总有更好的工具。举例来说，对于利用 MQTT 协议发送像视频或者图片那样的大型消息来说会给移动端应用程序带来不少问题。虽然 MQTT 擅长发送像应用状态数据那样的小型消息，但是在移动端或者嵌入式应用端上传视频或者图片时，HTTP 1.1 会是更理想的选择。在发送小型消息时，MQTT 比 HTTP 更胜一筹，但是在传送文件时，HTTP 协议则会更快。我们很容易忽略 HTTP 协议，但是它支持块文件上传，对于在相对不可靠的网络中传输大型媒体文件来说是绝佳选择。大多数成熟的客户端库会支持这一功能，你甚至无须动手编码来管理这些功能。如果使用 MQTT 的话，你就不得不亲力亲为了。

可变头部

在某些 MQTT 命令消息中的消息载荷是二进制打包数据，在称之为可变头部（variable headers）的数据结构中包含了消息的详细信息。不同的命令消息之间的格式差异很大。举例来说，CONNECT 消息的头部变量包含了连接协商的相关数据，而 PUBLISH 消息则包含了消息将要发往的 topic 以及唯一标识符。在 PUBLISH 命令消息中，消息载荷包含了头部变量以及不透明的应用层消息（见图 9.2）。

不同消息头中的值大小并不固定，例如 topic 名称，两字节的前缀用于表示值的大小（见图 9.3）。这种结构允许服务器端和客户端可以像从套接字里读取流数据那样读取和解码消息，而不用等待整条消息被完整读取后才开始解码。

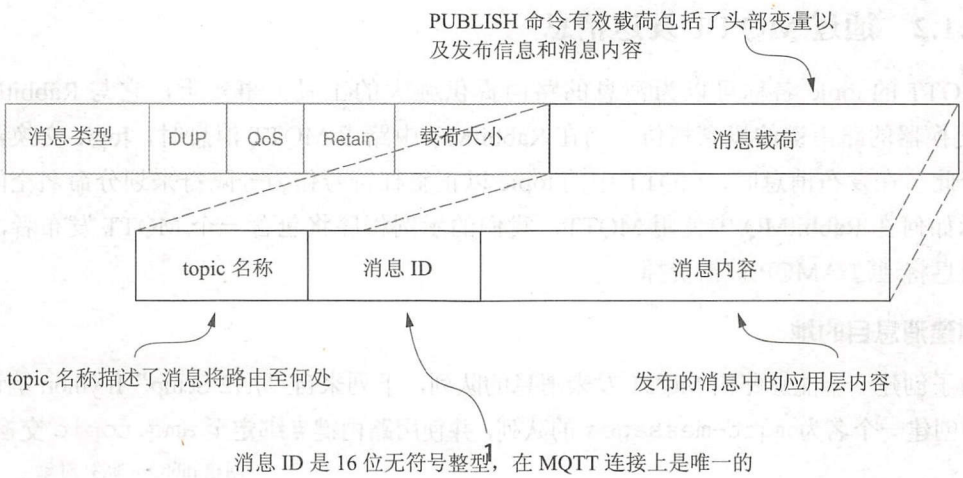


图 9.2　PUBLISH 命令消息的消息载荷

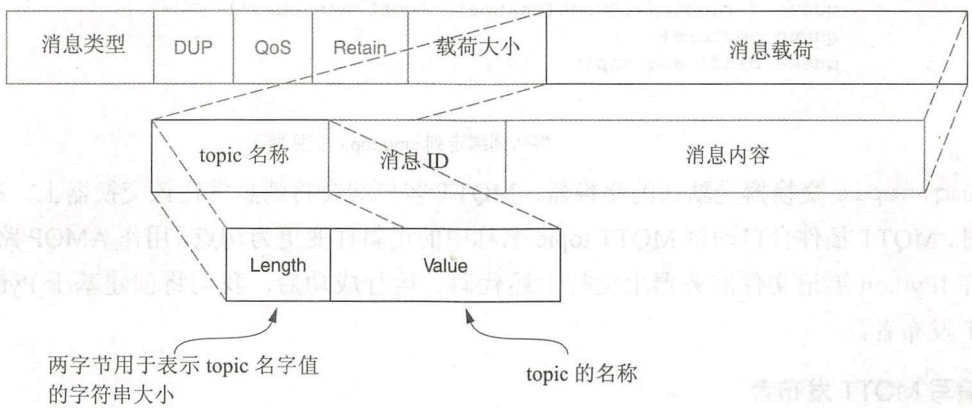


图 9.3　PUBLISH 命令消息头部变量中 topic 名称字段的结构

所有变量字段的值均为 UTF-8 编码，允许 32KB 的长度。重要的是，PUBLISH 消息中任何头部变量值会占用消息本身的最大大小。举例来说，如果将 topic 的名称设置为 `my/very/long/topic`，那么就会占用 23 个消息载荷可用字节，因此消息内容的总长度最多只能为 268,435,433 字节。

9.1.2 通过 MQTT 发送消息

MQTT 的 topic 名称可以为消息的路由提供强大的工具。事实上，它与 RabbitMQ 中 topic 交换器的路由键的概念相仿。当在 RabbitMQ 中路由 MQTT 消息时，topic 交换器就专门用于此。在发布消息时，MQTT 中的 topic 以正斜杠符号作为分隔符来划分命名空间。为了展示如何在 RabbitMQ 中使用 MQTT，我们的示例程序将包含一个 MQTT 发布者，它发布的消息将通过 AMQP 被消费掉。

创建消息目的地

为了创建一个能够路由 MQTT 发来消息的队列，下列来自“7.12 Setup”IPython 笔记文件示例将创建一个名为 mqtt-messages 的队列，并使用路由键 # 绑定至 amq.topic 交换器上。

```
import rabbitpy

with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        queue = rabbitpy.Queue(channel, 'mqtt-messages')
        queue.declare()
        queue.bind('amq.topic', '#')
```

创建 rabbitpy 队列对象

声明 mqtt-messages 队列

将队列绑定列 amq.topic 交换器上

amq.topic 交换器是默认的交换器，MQTT 客户端会将消息发往该交换器上。在发布消息时，MQTT 插件会自动将 MQTT topic 名称中的正斜杠变更为句点，用作 AMQP 路由键。

在 IPython 笔记文件服务器上运行上述代码。运行成功后，我们将创建基于 Python 的 MQTT 发布者。

编写 MQTT 发布者

mosquitto（https://pypi.python.org/pypi/ mosquitto）很受欢迎，可以选择用来实现通过 Python 和 MQTT 进行交互。它是一个异步库，专门用于运行异步 I/O 循环。不过我们将模拟一些内联操作，以便它和 RabbitMQ 进行通信。下列示例代码在“7.1.2 MQTT Publisher”笔记文件中。我们首先引入 mosquitto 库：

```
import mosquitto
```

在引入了库之后，我们将创建 mosquitto 客户端，并给客户端连接一个唯一的名字。在本例中我们就使用 rmqid-test 吧。在产品环境中，使用操作系统的进程 ID 来作为字符串的值是不错的选择：

```
client = mosquitto.Mosquitto('rmqid-test')
```

客户端类中有一个 connect 方法，可以向其中传入连接 MQTT 服务器的信息。connect 方法接收多个参数，包括 hostname、port 和 keepalive。在本例中，我们仅指定了主机名，port 和 keepalive 则使用了默认值。

```
client.connect('localhost')
```

若返回 0 时则代表连接成功。如果返回的值大于 0 时则代表连接服务器时发生了问题。

在连接成功后，现在可以通过客户端发送消息了。此处，我们传入 topic 的名称、消息内容以及数字 1 作为 QoS 的值，表示消息至少被发布一次，期望从 RabbitMQ 那里得到确认。

```
client.publish('mqtt/example', 'hello world from MQTT via Python', 1)
```

由于我们没有运行阻塞 I/O 循环，因此需要让客户端处理 I/O 事件。调用 client.loop() 方法来进行处理，该方法返回 0 代表成功：

```
client.loop()
```

现在可以从 RabbitMQ 断开连接，运行 client.loop 方法来处理 I/O 事件。

```
client.disconnect()
client.loop()
```

当运行这一程序之后，消息应当被成功发送出去，并处于之前声明的 mqtt-messages 队列中。让我们通过 rabbitpy 来验证一下吧。

通过 AMQP 获取从 MQTT 发布的消息

“7.1.2 Confirm MQTT Publish”笔记文件包含了下列代码，它使用 Basic.Get 从 mqtt-messages 队列获取消息，并用 Message.pprint() 方法打印消息的内容。

```
import rabbitpy

message = rabbitpy.get(queue_name='mqtt-messages')
if message:
    message.pprint(True)
    message.ack()
else:
    print('No message in queue')
```

使用 Basic.Get 从 RabbitMQ 获取消息

测试是否成功获取消息

确认消息

打印消息及其属性

如果没有收到消息的话，提示用户

当运行这段代码时，来自 RabbitMQ 的 AMQP 消息将被透明地从 MQTT 语义映射为 AMQP 语义。

```
Exchange: amq.topic
Routing Key: mqtt.example
```

```
Properties:
{'app_id': '',
 'cluster_id': '',
 'content_encoding': '',
 'content_type': '',
 'correlation_id': '',
 'delivery_mode': None,
 'expiration': '',
 'headers': {'x-mqtt-dup': False, 'x-mqtt-publish-qos': 1},
 'message_id': '',
 'message_type': '',
 'priority': None,
 'reply_to': '',
 'timestamp': None,
 'user_id': ''}
Body:
'hello world from MQTT via Python'
```

路由键不再是发送时的 topic 名称 mqtt/example，不过消息内容没有发生改变。RabbitMQ 将正斜杠替换为了句点，以匹配 topic 交换器的语义。同时，值得注意的是，AMQP 消息属性 headers 表中包含两个值 x-mqtt-dup 和 x-mqtt-publish-qos。这两个值包含了 MQTT 的 PUBLISH 消息头的值。

在验证了发布者之后，让我们来看看 MQTT 订阅者与 RabbitMQ 的异同吧。

9.1.3 MQTT 订阅者

当通过 MQTT 连接 RabbitMQ 来订阅消息时，RabbitMQ 将创建新的队列。队列名称将采用 mqtt-subscriber-[NAME]qos[N] 的格式。其中 [NAME] 是唯一的客户端名称，[N] 是客户端连接设置的 QoS 等级。举例来说，一个名为 mqtt-subscriber-facebookqos0 的队列，代表订阅者名称为 facebook，并且 QoS 设置为了 0。一旦为订阅请求创建队列之后，那么该队列将会采用 AMQP 点分路由键的语义，被绑定到 topic 交换器上。

订阅者能够使用类似于 AMQP 中 topic 和交换器间的路由键绑定的方式，通过字符串匹配或者模式匹配绑定到 topic 上。井号键（#）在 AMQP 和 MQTT 中可以用于多层匹配。不过对于 MQTT 客户端发送消息来说，加号符号（+）是用于路由键的单层匹配，而非星号（*）。举例来说，假设将图片消息通过 MQTT 发布至名为 image/new/profile 和 image/new/gallery 的 topic 上的话，订阅至 image/# 的消费者将能收到所有的图片消息，订阅至 image/new/+ 的消费者将能收到所有新的图片消息，而订阅至 image/new/profile 的消费者只能接收到新的资料图片消息。

下列来自“7.1.3 MQTT Subscriber”笔记文件的示例程序，将通过 MQTT 协议连接至

RabbitMQ，该程序将自己设置为订阅者，不断地循环直到收到消息为止。一旦接收到消息之后，它将取消订阅并从 RabbitMQ 断开连接。首先需要引入 mosquitto 和 os 库：

```
import mosquitto
import os
```

在创建 mosquitto 客户端时，可以使用 Python 的标准库 os 模块来获取订阅者的进程 ID，并使用它来作为唯一的 MQTT 客户端名称。在生产环境中需要更为随机或者健壮的订阅者命名方法以防止客户端重名，不过在这个示例程序中使用进程 ID 就足够了。

```
client = mosquitto.Mosquitto('Subscriber-%s' % os.getpid())
```

现在可以定义一些回调方法，这些回调方法会在 mosquitto 库执行的每个阶段被调用。首先，定义一个回调方法用于客户端连接时调用：

```
def on_connect(mosq, obj, rc):
    if rc == 0:
        print('Connected')
    else:
        print('Connection Error')
client.on_connect = on_connect
```

当 MQTT 消息被投递时，mosquitto 客户端将会调用回调方法 on_message。该回调方法将打印消息的信息，然后客户端将取消订阅。

```
def on_message(mosq, obj, msg):
    print('Topic: %s' % msg.topic)
    print('QoS: %s' % msg.qos)
    print('Retain: %s' % msg.retain)
    print('Payload: %s' % msg.payload)
    client.unsubscribe('mqtt/example')
client.on_message = on_message
```

最后的这个回调方法将在客户端取消订阅时调用，它将客户端从 RabbitMQ 断开连接。

```
def on_unsubscribe(mosq, obj, mid):
    print("Unsubscribe with mid %s received." % mid)
    client.disconnect()
client.on_unsubscribe = on_unsubscribe
```

在定义完成所有的回调方法之后，现在可以连接 RabbitMQ 并订阅 topic 了：

```
client.connect("127.0.0.1")
client.subscribe("mqtt/example", 0)
```

最后，通过执行 client.loop() 来调用 I/O 事件循环，并指定超时时间为 1 秒。下列

代码实现了上述功能，它会不断循环直到 `client.loop()` 不再返回 1 为止，这时客户端已从 RabbitMQ 断开连接。

```
while client.loop(timeout=1) == 0:
    pass
```

一旦打开笔记文件时，你可以通过单击 Cell 的下拉框，选择 Run All 来一次性运行所有的代码。单击“7.1.2 MQTT Publisher”页，选择 Cell>Run All，来发送新的消息。在订阅者页，你应当能看到如图 9.4 的输出。

正如你所见，点分形式的路由键已被转换回正斜杠划分的 topic 名字 `mqtt/example`。通过这种从 MQTT 的 topic 名称到 AMQP 路由键的双向转换，RabbitMQ 成功地以透明而自然的方式为每种客户端连接提供了桥接的方式。因此，RabbitMQ 不仅为 MQTT 创建了引人注目的平台，同时也创建了相比于特定协议的代理服务器更为健壮的消息通信平台。

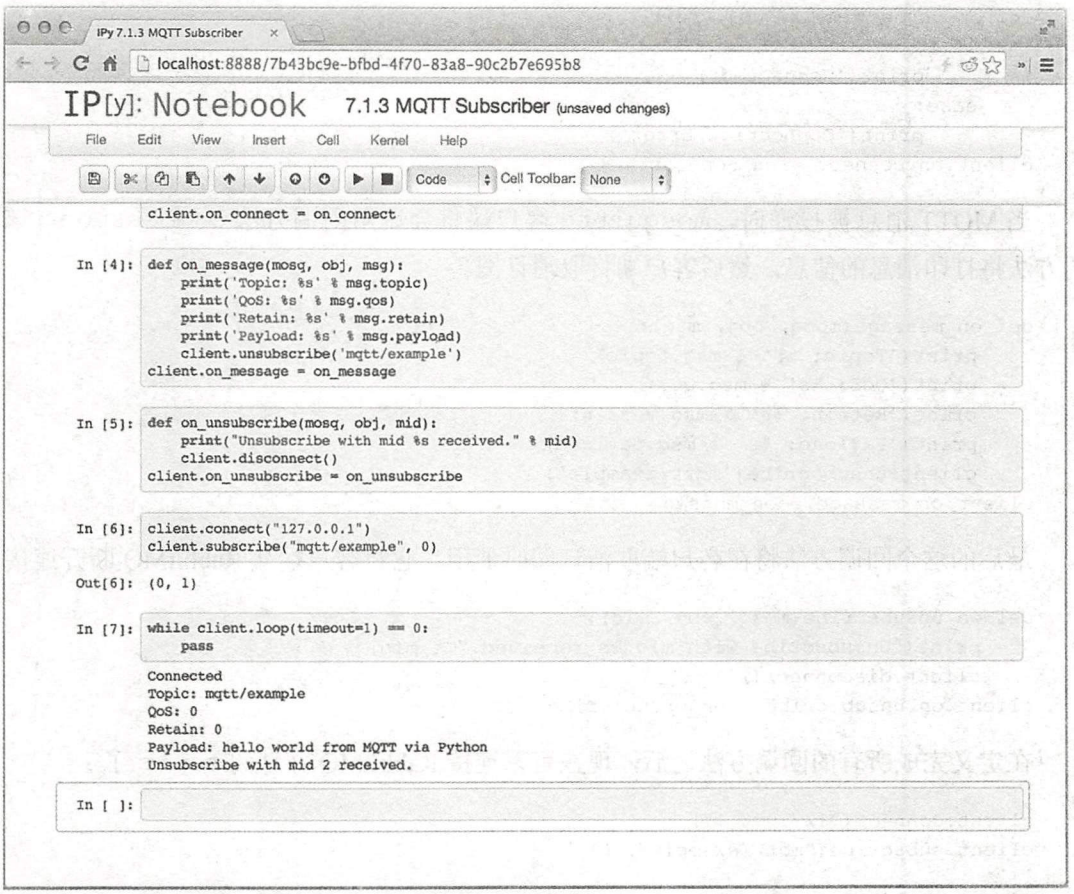


图 9.4 “MQTT Subscriber”IPython 笔记文件的输出

9.1.4　MQTT 插件配置

在介绍完 MQTT 的基本功能之后，你可能想要定制化 MQTT 的行为以便能够匹配 RabbitMQ 集群的方方面面，例如提供 MQTT 特定的认证凭据，或者订阅者的特定队列配置。为了更改配置，你需要编辑主 RabbitMQ 配置文件 rabbitmq.config。

在基于 UNIX 的操作系统中，RabbitMQ 配置文件通常位于 /etc/rabbitmq/rabbit.config。鉴于大多数配置文件使用数据序列化格式，rabbitmq.config 文件采用的是原生 Erlang 数据结构的代码格式。类似于 JSON 数组对象，RabbitMQ 配置包含了一个顶层的节代表 RabbitMQ 本身，然后每个需要配置的插件会有对应的节。在下列代码片段中，RabbitMQ 的 AMQP 监听端口设置成了 5672，MQTT 插件监听端口设置成了 1883。

```
[{rabbit,          [{tcp_listeners,  [5672]}]},
 {rabbitmq_mqtt, [{tcp_listeners,  [1883]}]}].
```

许多默认设置，例如虚拟主机和 MQTT 插件的默认用户名和密码，都借鉴了 RabbitMQ 的默认值。不同于 AMQP，MOTT 客户端无法选择虚拟主机。虽然这一行为可能会在将来的版本中变更，但是当前更改 MQTT 客户端使用的虚拟主机的唯一方式是修改 MQTT 配置节中采用 vhost 指令指定的默认值（正斜杠），将其修改为期望的值：

```
[{rabbitmq_mqtt, [{vhost, <<"/">}]}]
```

虽然 MQTT 提供了认证工具，但也有部分场景是无法满足的。对于这些场景来说，MQTT 插件的默认用户名和密码是 guest 和 guest 的组合。这两个默认值可以通过 default_user 和 default_pass 配置指令进行修改。如果想要对 MQTT 客户端开启认证的话，可以通过将 allow_anonymous 配置指令设置为 false 来禁止默认的用户行为。

提示　MQTT 应用架构可能会需要针对不同类型的 MQTT 客户端采用不同的配置。采用 RabbitMQ 集群是解决单一虚拟主机和默认用户名密码限制的解决方案之一。通过每个节点不同配置的方式，在 RabbitMQ 集群内共享 MQTT 消息的同时，集群中每个节点启用不同的默认配置来接收 MQTT 连接。RabbitMQ 集群节点间无需统一的配置。

表 9.1 描述了每条 MQTT 插件配置指令和对应的默认值。这些值直接影响 MQTT 插件的行为、MQTT 客户端和消息路由。

表 9.1 MQTT 插件配置选项

指 令	类 型	描 述	默认值
allow_anonymous	Boolean	开启 MQTT 客户端无需认证即可连接	true
default_user	String	当 MQTT 客户端没有提供认证凭证时使用的用户名	guest
default_password	String	当 MQTT 客户端没有提供认证凭证时使用的密码	guest
exchange	String	当发布 MQTT 消息时使用的 topic 交换器	amq.topic
Prefetch	Integer	为 MQTT 监听队列设置 AMQP QoS 的预取数量	10
ssl_listeners	Array	在 SSL 连接上监听 MQTT 的 TCP 端口设置。如果指定的话，那么配置文件中的顶层 rabbit 节必须包含 ssl_options 配置节	[]
subscription_ttl	Integer	在订阅者预料之外断开连接后，保存订阅队列的市场，单位为毫秒	1800000
tcp_listeners	Array	用于监听 MQTT 连接的 TCP 端口	1833
tcp_listen_options	Array	配置指令数组，用于修改 MQTT 插件的 TCP 行为	见表 9.2

像 exchange、prefetch 和 vhost 等这些指令更有可能针对特定的环境进行修改，而像 tcp_listen_options 调整起来要格外小心。

表 9.2 描述了 RabbitMQ 文档中包含的 tcp_listen_options 指令，以及在 MQTT 客户端和 MQTT 插件中这些指令的调整对 TCP 连接产生的作用。这些值是 Erlang TCP API 中用来调整 TCP 套接字的子集。如果想要了解其他指令详情，请参考 http://erlang.org/doc/man/ gen_tcp.html 中的 gen_tcp 文档。根据 RabbitMQ 配置的工作方式，配置文件中指定的值会以 listen_option 参数形式，透明地传递给 Erlang gen_tcp:start_link。大多数情况下，MQTT 插件的默认值都不应当作调整。RabbitMQ 团队对这些值做了充分的测试和优化。

表 9.2　MQTT 插件的 tcp_listen_options

指　令	类　型	描　述	默认值
binary	Atom	表示套接字是否是二进制 TCP 套接字。不要删除	N/A
packet	Atom	用于调整 Erlang 内核在提交给 RabbitMQ 之前处理 TCP 数据的方式。详情请参看 gen_tcp 文档	raw
Reuseaddr	Boolean	指示操作系统允许 RabbitMQ 在需要时重用监听端口，即使该套接字处于忙碌中。	true
backlog	Integer	指定在重用新连接前，等待中的客户链接数量。等待的客户端连接是新的 TCP 套接字连接，RabbitMQ 尚未处理	10
nodelay	Boolean	指定 TCP 套接字是否启用 Nagle 算法，等待汇总底层 TCP 数据以实现更高效的数据传输。默认值是 true，RabbitMQ 会在需要时发送 TCP 数据，大多数情况下 MQTT 消息通信会更为快速，而非缓存小型消息包再分组一起发送	true

让我们回顾一下，在不断发展的移动计算和嵌入式设备领域中，MQTT 是一款强大的轻量级消息通信工具。如果你将 RabbitMQ 作为消息通信架构（包括移动设备）的核心的话，那么强烈建议你考虑使用 MQTT 和 RabbitMQ MQTT 插件。它不仅实现了 MQTT 语义和 RabbitMQ 的 AMQP 之间透明转换，而且简化了统一消息总线的开发。虽然配置上的不足导致单个节点上无法实现复杂的 MQTT 生态，但是扩展 MQTT 插件以提供动态虚拟主机和交换器的工作正在进行当中。同时，集群中的多个 RabbitMQ 节点可以创建出复杂的 MQTT 拓扑结构。

假如消息通信架构无法从 MQTT 中受益，但是你仍倾向于较 AMQP 更为轻量级的解决方案来实现与 RabbitMQ 之间的通信的话，那么也许 STOMP 会合适。

9.2　STOMP 和 RabbitMQ

STOMP 的前身是 TMPP。后来 Brian McCallister 于 2005 年首次将其明确为面向文本的消息协议（STOMP）。由于参照 HTTP 进行建模，STOMP 成为了简单易读的基于文本的协议。该协议最开始在 Apache ActiveMQ 中进行了实现。由于奉行简单的原则，STOMP 现在支持了多种消息代理服务器实现，并且有大多数流行的编程语言的客户端实现。

STOMP 1.2 规范于 2012 年发布。RabbitMQ 支持这一版本（以及上一版本）的 STOMP 规范。STOMP 作为一个插件随 RabbitMQ 核心包的一部分进行发布。像 AMQP 和 MQTT 那样，

对 STOMP 协议的理解有助于你在应用开发过程中建立起自己的观点。

9.2.1 STOMP 协议

STOMP 专门设计用于基于流的处理，STOMP 帧是 UTF-8 文本，由命令和命令对应的载荷组成，并以 null（0x00）字节结束。不同于 AMQP 和 MQTT 协议，STOMP 是可读的，而且不需要二进制位封装信息来定义 STOMP 消息帧和内容。

举例来说，下列代码片段是一个 STOMP 帧，用于连接至消息代理服务器。它使用 ^@（ASCII 表示为 control-@）来表示帧的结尾 null 字节。

```
CONNECT
accept-version:1.2
host:rabbitmq-node

^@
```

在本示例中，CONNECT 命令告诉接收到消息的代理服务器，客户端想要发起连接。之后的两个头字段 `accept-version` 和 `host`，提示代理服务器客户端想要协商的连接。最后，一个空行和 null 字节，表示 CONNECT 帧的结束。

如果请求成功的话，那么代理服务器将会向客户端返回 CONNECTED 帧。该帧与 CONNECT 帧非常类似：

```
CONNECTED
version:1.2

^@
```

类似于 AMQP，STOMP 命令对于消息代理服务器来说就是 RPC 风格的请求，当中的某些命令会向客户端发送响应。STOMP 命令的标准集包含的概念类似于 AMQP 和 MQTT，包括连接协商、消息发布，以及从消息代理服务器上订阅并接受消息。如果你需要了解更多有关协议的内容，请访问 https://stomp.github.io/ 以了解 STOMP 协议的规范。

为了演示如何在 RabbitMQ 中使用 STOMP，让我们从简单的消息发布者开始吧。

9.2.2　发布消息

当使用 STOMP 来发布消息时，它采用通用概念目的地来描述消息将被发往何处。在 RabbitMQ 中，STOMP 目的地取自下列之一：

- 当消息被发布时或者当客户端发送订阅请求时，STOMP 插件自动创建的队列。
- 通过普通的方式创建的队列，例如通过 AMQP 客户端创建的，或者通过管理 API 创建。
- 交换器和路由键的组合。
- 使用 STOMP 的 topic 目的地自动映射而成的 `amq.topic` 交换器。
- 当在 STOMP 命令 SEND 中使用了 `reply-to` 头时的临时队列。

这些目的地都采用了正斜杠进行分隔。这种分隔能分离出目的地类型。大多数情况下，还能分离出交换器、路由键或者队列信息等。

为了演示消息目的地的使用方法以及如何发布消息，让我们使用 Python 库 stomp.py 向 STOMP 队列发送一条消息。

注意　STOMP 插件的工作机制类似于转换器或者说是 RabbitMQ 自己的代理层。因此，它像 AMQP 客户端那样工作，创建 AMQP 信道并向 RabbitMQ 自己发送 AMQP RPC 请求。STOMP 发布者像其他 AMQP 发布者那样同样受限于速率限制和链接阻塞，而且 STOMP 协议中没有定义语义，让发布者知道自己被阻塞或限流了。

向 STOMP 定义的队列发送消息

通过 STOMP 发送消息和通过 MQTT 或者 AMQP 发送消息类似。想要直接向队列发送消息的话，采用格式为 `/queue/<queue-name>` 的字符串作为目的地。

在下面这个例子中，我们将采用目的地 `/queue/stomp-messages`。向该目的地发送消息的话，就会采用 RabbitMQ 默认交换器行为方式向 `stomp-messages` 队列发布消息。如果队列不存在的话，那么就会自动创建。示例代码存放在“7.2.2 STOMP Publisher”笔记文件中。

```
import stomp
conn = stomp.Connection()
conn.start()
conn.connect()
conn.send(body='Example Message', destination='/queue/stomp-messages')
conn.disconnect()
```

当 STOMP 插件创建队列时，它会在内部以默认参数值发起 RPC 请求 `Queue.Declare`。这意味着如果你在 RabbitMQ 服务器上已经使用默认值创建了已存在的队列的话，你仍然可以使用 STOMP 队列目的地发布消息。如果队列使用了消息 TTL 或者其他自定义参数创建的话，你就需要改用 AMQP 定义的队列目的地了。

向 AMQP 定义的队列发送消息

根据 STOMP 的 RabbitMQ 实现，STOMP 插件扩展了目的地预发，允许向自定义设置的 AMQP 定义的队列发布消息。为了实现上述功能，首先需要使用“7.2.2 Queue Declare”笔记文件中的 rabbitpy 库创建拥有最大长度的队列。

```
import rabbitpy
with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        queue = rabbitpy.Queue(channel, 'custom-queue',
                               arguments={'x-max-length': 10})
        queue.declare()
```

队列创建之后，需要使用 AMQP 定义的队列目的地语法。通过创建 /amq/queue/<queue-name> 格式的目的地字符串，STOMP 插件才能将消息路由至 custom-queue 队列。示例代码位于“7.2.2 Custom Queue”笔记文件中。

```
import stomp
conn = stomp.Connection()
conn.start()
conn.connect()
conn.send(body='Example Message', destination='/amq/queue/custom-queue')
conn.disconnect()
```

直接向队列发送消息带来的一个问题是不能享受到使用功能丰富的交换器和路由键来接收 AMQP 消息所带来的好处了。幸运的是，STOMP 插件允许使用特别的格式化目的地字符串来实现上述目的。

向交换器发送消息

通过采用 /exchange/<exchange-name>/<routing-key> 格式的方式，我们可以在 RabbitMQ 的 STOMP 插件中使用路由键向交换器发送消息。这种做法使得我们通过 STOMP 发布消息的同时能够享受到 AMQP 那样的灵活性。

下列来自“7.2.2 Exchange and Queue Declare”笔记文件的示例代码会将参数都设置好，以便向自定义交换器发布 STOMP 消息。

```
import rabbitpy
with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        exchange = rabbitpy.Exchange(channel, 'stomp-routing')
        exchange.declare()
        queue = rabbitpy.Queue(channel, 'bound-queue',
                               arguments={'x-max-length': 10})
        queue.declare()
        queue.bind(exchange, 'example')
```

我们声明了交换器和队列，并将队列绑定到了交换器上，现在是时候向这个新队列发布消息了。下列代码示例来自“7.2.2 Exchange Publishing”笔记文件。

```
import stomp
conn = stomp.Connection()
conn.start()
conn.connect()
conn.send(body='Example Message',
          destination='/exchange/stomp-routing/example')
conn.disconnect()
```

上述示例程序中交换器路由的灵活性显而易见。不过，无需声明交换器或者使用更长的交换器目的地字符串，也能从 topic 交换器路由的灵活性中受益。在发送消息时，使用 STOMP topic 目的地字符串即可。

向 STOMP TOPIC 交换器发送消息

Topic 目的地字符串类似于队列目的地字符串。它使用的普通格式可以被所有支持 STOMP 协议的代理服务器识别。通过将目的地字符串格式化为 /topic/<routing-key> 的形式，那些经由 STOMP 发往 RabbitMQ 的消息将会通过 amq.topic 交换器发往所有绑定至路由键的队列。

不用创建新的队列，你可以将先前创建的 bound-queue 队列，采用 # 作为路由键，绑定至 amq.topic 交换器上，以便接收所有发往该交换器上的消息。下列示例代码位于“7.2.2 Bind Topic”笔记文件中。

```
import rabbitpy
with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        queue = rabbitpy.Queue(channel, 'bound-queue')
        queue.bind('amq.topic', '#')
```

在绑定了队列之后，现在可以通过 STOMP 协议向 /topic/routing.key 路由键发送消息了。该示例程序位于“7.2.2 Topic Publishing”笔记文件中。

```
import stomp
conn = stomp.Connection()
conn.start()
onn.connect()
conn.send(body='Example Message',
          destination='/exchange/stomp-routing/example')
conn.disconnect()
```

大部分消息发布的场景均有所涉及，包括队列、amq 队列、交换器以及 topic 目的地字符串。除此之外，STOMP 还添加了一个很棒的功能，它重现了第 6 章中的部分工作。如果

你通过 STOMP 发布消息，并设置 reply-to 消息头的话，那么 STOMP 插件将会自动创建 RPC 应答队列用于消息消费。

使用临时应答队列

当消息通信架构中需要支持发布者和消费者之间的 RPC 调用时，STOMP 提供了内建于 RabbitMQ STOMP 插件的便利功能。通过在发送 STOMP 消息时设置 reply-to 消息头，RabbitMQ 会自动创建响应队列，并且会设置它的排他标志和自动删除标志，以便确保消息发布的 STOMP 连接是唯一可以从应答队列消费消息的连接。此外，一旦消息发布应用断开的话，那么该应答队列将会自动从 RabbitMQ 中删除。

下列来自“7.2.2 Reply-To”笔记文件的示例代码演示了如何设置 reply-to 消息头。我们将在本章的下个小节中讲解如何通过 STOMP 消费消息。那么现在，我们让应答队列在消息发布后自动删除。

```
import stomp
conn = stomp.Connection()
conn.start()
conn.connect()
conn.send(body='Example Message',
          destination='/exchange/stomp-routing/example',
          headers={'reply-to': 'my-reply-queue'})
conn.disconnect()
```

设置 reply-to 消息头的主要目的是 STOMP 消息可以具有任意的消息头。这些消息头与 AMQP 消息属性最为类似。

STOMP 上的 AMQP 消息属性

你可以利用 STOMP 消息头向消息代理服务器传入任意的消息头。这一功能为消息发布者开启了应答队列的自动创建的 reply-to 功能。虽然任意消息头对应用程序来说很有用，但是如果考虑到那种同时使用 STOMP 和 AMQP 两种混合协议场景的话，那么我们考虑使用在两种协议中都可用的受限消息头集合。如果使用的消息头匹配了 AMQP 消息属性名称的话，那么 STOMP 插件会自动将消息头的值映射到 AMQP 消息属性中去。

下列来自“7.2.2 Send with Message Headers”笔记文件的示例程序设置了消息头。这些消息头将会被转换成 AMQP 消息属性。

```
import stomp
import time
conn = stomp.Connection()
conn.start()
conn.connect()
```

```
conn.send(body='Example message with Headers',
          destination='/queue/stomp-messages',
          headers={'app-id': '7.2.2 Example',
                   'priority': 5,
                   'reply-to': 'reply-to-example',
                   'timestamp': int(time.time())})
conn.disconnect()
```

如果设置的 STOMP 消息头无法匹配 AMQP 消息属性的话，那么这些值会被填充到 AMQP 的 headers 消息属性当中去。正如预期的那样，消息中填充的那些 AMQP 消息属性均来自 STOMP 消息头。

只有一个例外，那就是 AMQP 消息属性 message-id。作为 STOMP 协议的一部分，该值会被自动设置。同时，在使用 STOMP 协议向 RabbitMQ 发送消息时不应该手动设置该值。

使用 STOMP 协议向 RabbitMQ 发布消息要比使用 AMQP 协议多做一点工作，但是这同时也带来了不少好处，例如自动队列创建和自动创建 reply-to 队列。通过使用不同的目的地字符串格式，你可以像 AMQP 协议那样，将消息直接发往队列或是交换器中。STOMP 插件会自动将 AMQP 语义映射为 STOMP 消息，反之亦然。

你可以通过消费之前示例程序发布的消息来实际验证这一自动映射机制。在下一节中，我们将通过 STOMP 订阅者来消费消息，包括那些含有消息头的消息。

9.2.3　消费消息

与 MQTT 类似，STOMP 客户端更像是订阅者而非消费者。不过，由于 RabbitMQ 本质上来讲是一个 AMQP 代理服务器，STOMP 插件把 STOMP 订阅者看成是 AMQP 消费者那样从 RabbitMQ 队列中获取消息。当你通过 STOMP 进行订阅时，大多数情况下会创建一个队列用于消息的消费。

STOMP 插件中较为出彩的功能是所有用于通过 STOMP 发送消息目的地字符串类型都可以在 STOMP 中用来订阅消息。本节将介绍在 STOMP 订阅者中如何利用目的地字符串实现以下功能：

- 从自动创建的队列中消费消息。
- 从先前定义的 AMQP 队列中消费消息。
- 通过订阅到交换器上来消费消息。
- 通过订阅值 STOMP topic 来消费消息。

当通过 STOMP 发送消息时，你可以设置 reply-to 消息头，它将为 STOMP 连接自动创建一个排他的自动删除的队列。相对的，消费 reply-to 消息的处理方式和从订阅至 STOMP 定义的队列消费消息的方式是一致的。

首先，让我们订阅上一节中使用的 stomp-messages 队列。

订阅至 STOMP 定义的队列

当使用 /queue/<queue-name> 格式的目的地字符串发布消息时，RabbitMQ 会创建 STOMP 定义的队列。前一节中，我们在“7.2.2 Stomp Publisher”中向 RabbitMQ 发布的消息使用了 STOMP 的 send 命令。下列示例代码来自“7.2.3 Queue Subscriber”笔记文件。示例代码会消费消息，打印出每条消息的消息体。由于订阅者代码有点复杂，让我们一步步分析吧。

首先，在运行订阅者代码之前需要引入必须的 Python 库：

```
import stomp
import pprint
import time
```

Python 库 stomp.py 需要 ConnectionListener 对象处理从消息代理服务器接收到的消息。该对象应当包含 on_message 方法，在订阅者接收到消息时调用。本示例代码将打印出消息头和消息体。此外，程序一旦接收到一条消息后就会退出。在示例监听器中添加的标识符 can_stop 用于判断示例代码何时停止。

```
class Listener(stomp.ConnectionListener):
    can_stop = False
    def on_message(self, headers, message):
        if headers:
            print('\nHeaders:\n')
            pprint.pprint(headers)
        print('\nMessage Body:\n')
        print(message)
        self.can_stop = True
```

在定义了监听器后，现在可以创建 STOMP 连接对象，设置连接监听器对象，开启连接并连接至 RabbitMQ：

```
listener = Listener()
conn = stomp.Connection()
conn.set_listener('', listener)
conn.start()
conn.connect()
```

一旦建立连接之后，Connection.subscribe 方法会发送 STOMP 订阅请求给 RabbitMQ，而 RabbitMQ 会自动确认接收到的消息。

```
conn.subscribe('/queue/stomp-messages', id=1, ack='auto')
```

为了确保代码在接收到消息前一直处于等待状态，此处会使用循环，每次会 sleep 一秒钟，直到 `Listener.can_stop` 被设置为 `True`：

```
while not listener.can_stop:
    time.sleep(1)
```

最后，一旦接收到消息之后，连接将会断开：

```
conn.disconnect()
```

运行“7.2.2 Send with Message Headers”笔记文件中的代码，直到接收到定义了消息头的消息为止。接收到的消息输出类似于图 9.5 IPython 笔记文件上的输出。

值得注意的是，AMQP 消息属性被合并到了 STOMP 消息头中去了，包括 `content-length` 和 `destination`。接收到的消息中的任何 AMQP 消息属性值都会被拉平塞入 STOMP 消息的头中。

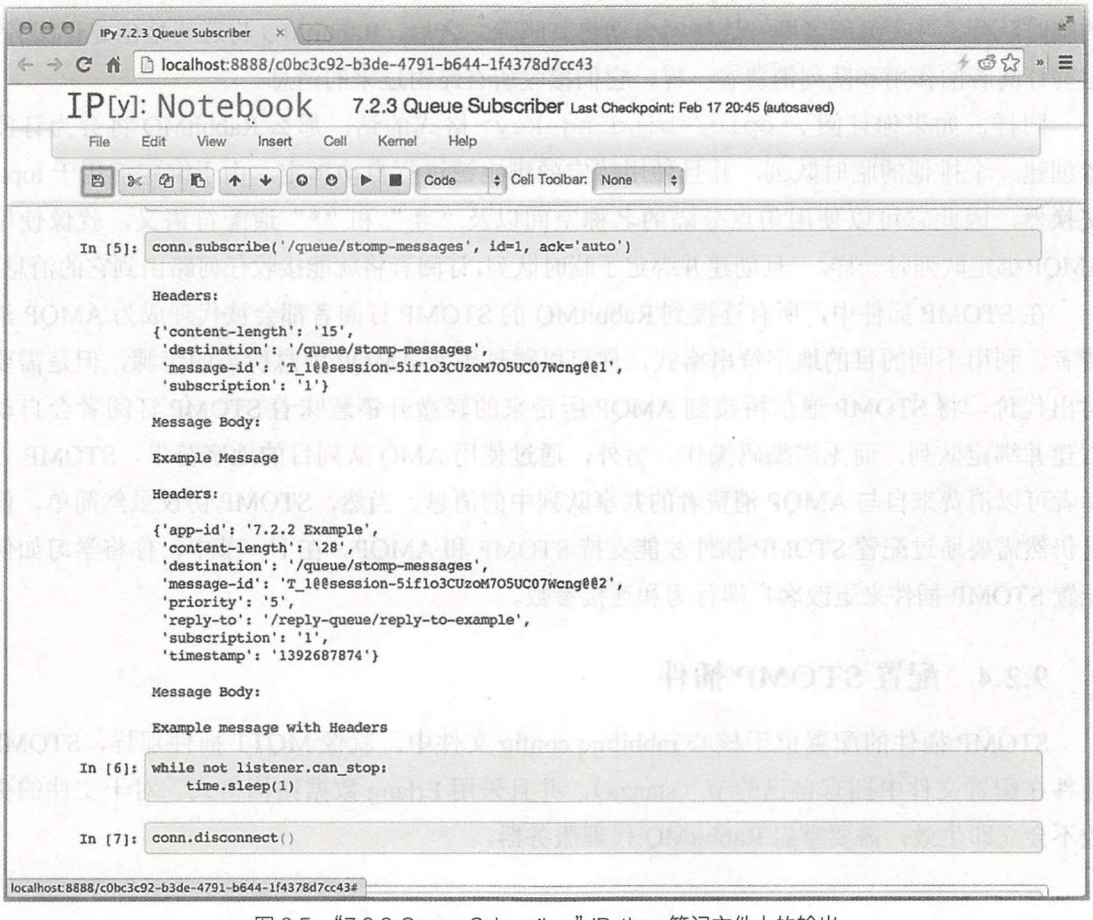


图 9.5　“7.2.3 Queue Subscriber”IPython 笔记文件上的输出

就像在发送 STOMP 消息那样，如果你将消息发送至使用自定义参数声明的 AMQP 队列的话，那么订阅将会失败，并且你将不会收到任何消息。幸运的是，RabbitMQ 团队针对这种情况添加了 AMQ 队列目的地字符串格式。

订阅 AMQP 定义的队列

如果你需要混合使用 STOMP 订阅者和 AMQP 消费者，使用自定义参数从同一个队列消费消息，又或者你需要单个 STOMP 订阅者使用自定义参数从队列中接收消息的话，那么你可以在使用 STOMP 订阅者时使用 `/amq/queue/<queue-name>` 这一目的地格式。

订阅交换器或者 TOPIC

STOMP 插件的另一个出彩的功能是它可以让你订阅格式为 `/exchange/<exchange-name>/<binding-key>` 的交换器。当你这样做时，RabbitMQ 将为订阅者创建一个排他的临时队列，并在订阅者断开连接时自动将其删除。之后，RabbitMQ 将透明地创建订阅者。这些订阅者的作用和队列消费者一样，它们接收所有路由过来的消息。

同样，如果你订阅 `/topic/<binding-key>` 格式的话，那么 RabbitMQ 将会为订阅者创建一个排他的临时队列，并且使用特定的绑定键进行自动绑定。由于绑定键用于 topic 交换器，因此它可以使用句点分隔的名称空间以及“#”和“*”通配符语义，就像使用 AMQP 绑定队列时一样。一旦创建并绑定了临时队列，订阅者将就能接收任何路由到它的消息。

在 STOMP 插件中，所有连接到 RabbitMQ 的 STOMP 订阅者都会被代理成为 AMQP 消费者。利用不同的目的地字符串格式，你可以跳过消费 AMQP 消息所需的步骤，但是需要付出代价。将 STOMP 通信桥接到 AMQP 所带来的轻微开销意味着 STOMP 订阅者会自动创建并绑定队列，而无需编码操作。另外，通过使用 AMQ 队列目的地字符串，STOMP 订阅者可以消费来自与 AMQP 消费者的共享队列中的消息。当然，STOMP 协议虽然简单，但是仍然需要通过配置 STOMP 插件才能支持 STOMP 和 AMQP。在下一节中，你将学习如何配置 STOMP 插件来更改客户端行为和连接参数。

9.2.4 配置 STOMP 插件

STOMP 插件的配置位于核心 rabbitmq.config 文件中。就像 MQTT 插件那样，STOMP 插件在配置文件中拥有自己的节（stanza），并且采用 Erlang 数据结构格式。对于文件的更改不会立即生效，需要重启 RabbitMQ 代理服务器。

就像下列代码片段所展示的那样，顶层 STOMP 插件配置位于 rabbitmq_stomp 配置部分。

```
[{rabbit,          [{tcp_listeners,  [5672]}]},
 {rabbitmq_stomp, [{tcp_listeners,  [61613]}]}].
```

表 9.3 展示了 STOMP 插件的配置选项细节。

表 9.3　STOMP 插件配置选项

指　令	类　型	描　述	默认值
default_user	String	当客户端没有提供认证凭据时使用的用户名	[{login, "guest", passcode, "guest"}]
implicit_connect	Integer	允许 STOMP 连接不发送 CONNECT 帧。开启之后，CONNECTED 帧将不会被发送	False
ssl_listeners	Array	在 SSL 连接上用于监听 STOMP 的 TCP 端口。如果指定该值的话，那么配置文件中顶层 rabbit 节必须包含 ssl_options 配置节	[]
ssl_cert_login	Boolean	允许基于 SSL 证书的认证方式	False
tcp_listeners	Array	用于监听 STOMP 连接的 TCP 端口	[61613]

9.2.5　在 Web 浏览器中使用 STOMP

与 RabbitMQ 绑定发布的还有 Web STOMP 插件。作为 RabbitMQ 上专门的扩展插件，Web STOMP 采用 SockJS 库，添加了 websocket 兼容的 HTTP 服务器。这使得用户可以直接使用 Web 浏览器与 RabbitMQ 进行交互。默认情况下，Web STOMP 插件监听 15670 端口，并且除了 STOMP 心跳功能外，完整支持 STOMP 协议。这是由于 Web STOMP 和 RabbitMQ 进行通信所采用的 SockJS 的性质决定了无法使用心跳。

Vagrant 虚拟机启用了 Web STOMP，并包含相关的使用示例。要查看 Web STOMP 库和服务的丰富的示例，请访问 http://localhost:15670/web-stomp-examples/。

在完整理解 Web STOMP 并将其作为应用程序的解决方案之前，若将 RabbitMQ 服务器向互联网公开，就像使用任何其他应用程序或服务一样的话，带来的安全隐患值得深思。将 RabbitMQ Web STOMP 服务器作为独立群集或服务器，并使用像 Shovel 和 Federation 插

件等工具将这些服务器桥接到主服务器上。这样一来可以减轻恶意客户端产生的影响，具有重要意义。有关 Web STOMP 的更多信息，请访问 www.rabbitmq.com/web-stomp.html 上的插件页面。

STOMP 协议是一种人类可读的基于文本的流协议，其设计简单易用。尽管 AMQP 和 MQTT 等二进制协议可能更高效，STOMP 协议通过使用更少的数据来传输相同的消息也颇具优势，特别是在使用 STOMP 插件和 RabbitMQ 时。队列创建和绑定行为需要较少的代码，但也需要付出代价。由 STOMP 插件创建的代理 AMQP 连接，在与 RabbitMQ 通信进行时需要对 STOMP 数据进行翻译，这相对于直接使用 AMQP 连接来说会有额外的开销。

由于发布和使用 AMQP 消息时有各种各样选项可以进行配置，我强烈建议你在产品环境中使用 STOMP 和 RabbitMQ 之前对其进行基准测试。每种协议都各有其优缺点，需要依使用场景而定。

在下一节中，我们将介绍 statelessd。它是一款高性能、无状态的 Web 应用程序，可以将消息发布到 RabbitMQ 中。它适用于专优化那些单一事务、发后即忘的场景，解决使用 AMQP 和 STOMP 协议带来的额外开销。

9.3 通过 HTTP 进行无状态发布

在某些场景中，AMQP、MQTT、STOMP 以及其他有状态协议对于无法维持到 RabbitMQ 的长久连接的高速消息通信来说十分昂贵。这些协议在进行消息相关的操作之前会有一些连接开销。从性能的角度来看，相对于短连接来说，它们可能不太理想。正是认识到了这一点才导致开发了 statelessd。它是一款 HTTP 转 AMQP 的消息发布代理。对于客户端应用程序来说，它提供了高性能、发后即忘的消息发布机制，而无需受困于连接状态管理开销。

9.3.1 statelessd 的由来

在 2008 年年中，我们开始在 MeetMe.com（然后是 myYearbook.com）构建异步消息通信架构，将数据库写入操作与基于 PHP 的 Web 应用程序进行分离。最初，我们使用 Apache ActiveMQ 来构建架构。ActiveMQ 是一个基于 Java 的消息代理服务，支持 STOMP 协议。就像 memcached 那样能够在根本上对数据库读取操作进行完美地扩展，消息通信、STOMP 协议以及 ActiveMQ 使得我们可以创建消费者应用程序。这从根本上改变了我们的思考方式，让我们重新思考数据库写入操作、如何限定工作负载以及如何扩展昂贵的计算工作负载。

随着流量的不断增长，我们遇到了 ActiveMQ 的扩展问题，并开始评估其他消息代理服务器。当时，RabbitMQ 展示了很多诱人的承诺并同样支持我们在 ActiveMQ 中使用的 STOMP 协议。在我们迁移到 RabbitMQ 时，我们发现它对我们的环境来说是一个不错的选择，但同时也引入了新的问题。

当我们开始使用 RabbitMQ 时，我们立刻发现对于我们的 PHP 应用程序栈来说，有状态的 AMQ 协议非常昂贵。我们发现 PHP 无法维持跨客户端请求的开放连接和信道状态。为了发布消息，PHP 应用程序在处理每个请求时，都需要与 RabbitMQ 建立新的连接。

不要误解我的意思，创建与 RabbitMQ 的 AMQP 连接真正所需的时间并不多，以毫秒为单位进行计量。但是，当每秒发布数以万计的消息时，通常每个 Web 请求会发送一次或两次消息，因此每秒将会向 RabbitMQ 发起数以万计的连接。为了解决这个问题，我们最终创建了 HTTP 转 AMQP 的发布网关 statelessd。它需要接收高速的 HTTP 请求，同时管理用于消息发布所需的连接栈。另外，它不会成为性能瓶颈，并且能可靠地将消息发送到 RabbitMQ。

在将 statelessd 作为开源软件发布之后，我们发现我们并不是唯一面对这个问题的人。在 2013 年，来自 Weebly 的同行创建了一个名为 Hare 的 statelessd 克隆 (https://github.com/Weebly/Hare)，它是用 Go 语言编写的。

9.3.2　使用 statelessd

Statelessd 在设计上旨在尽可能地减少开销。它期望通过其发布 HTTP 消息的客户端使用本地 HTTP 约定来将所有必须的信息传达给本机 AMQP 消息。HTTP URI 中路径的第一部分包含 RabbitMQ 中用于发布消息的虚拟主机。另外，交换器和路由键也作为路径组件包含在请求中：

```
http://host[:port]/<virtual-host>/<exchange>/<routing-key>
```

使用 HTTP 基本身份验证头来验证用户名和密码。当一个请求进入时，statelessd 守护进程会查看 RabbitMQ 用户名、密码和虚拟主机的组合是否存在于其开放连接栈中。如果存在，守护进程将使用该开放连接发布消息，向客户端返回“请求已处理，不返回内容”（204）状态。

由于 statelessd 通常运行在受控环境中，因此由于认证导致的问题非常少见，因此需要设计一个折衷方案以实现最佳请求效率。如果连接未建立，statelessd 将在内部缓冲消息，启动一个异步进程以连接到 RabbitMQ，并向客户端返回 204 状态。一旦连接建立之后，所有缓冲的特定证书组合消息将被发送出去。如果连接出现问题，凭证的组合将被标记为有问

题，同时随后的任何请求都将收到 424 信息，即“由于先前请求的性质而导致请求失败”错误。

Statelessd 请求使用 HTTP POST 来发送标准的表单编码的键 / 值对，其中携带了要发布的消息体和属性。Statelessd 请求的有效键值包括消息体、实际消息体本身的值和标准 AMQP 消息属性名称。其中破折号字符被替换为下画线。例如，如果要设置 `message-id` 属性，请求的有效内容中应包含指派给 `message_id` 键的值。有关 Statelessd 请求负载中有效键的完整列表，请参阅 https://github.com/ gmr/statelessd 上的文档。

9.3.3 运营架构

Statelessd 的设计理念决定了它应当与消息发往的 RabbitMQ 服务器运行在同一台服务器上。它会运行一个基于 Python 的守护进程，通常为服务器上的每个 CPU 核心都配置一个后端进程。每个后端进程都有自己的 HTTP 端口。这些处理流程是聚合在一起的，并且可以使用像 Nginx 这样的反向代理服务器（见图 9.6）通过单个端口进行代理，从而提供了一个横向扩展解决方案。基于这一解决方案的基准测试显示每台服务器每秒钟可以处理数十万条消息。

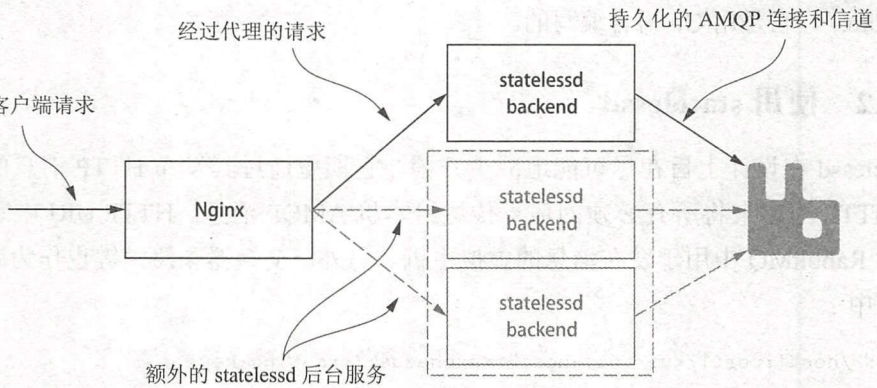


图 9.6 Statelessd 运营架构

如果需要在多台服务器上运行无状态服务器的话，那么可以将所有服务器的上 Nginx 实例中添加到负载平衡器中，以便在集群中的多个服务器之间分配消息发布请求。Statelessd 包含一个用于收集统计数据的 URL 终端，可用于比较 statelessd 节点群和 RabbitMQ 服务器两者之间的消息吞吐率。有关安装和配置 statelessd 的信息，请参阅 https://github.com/gmr/statelessd 上的 statelessd 文档。

9.3.4　通过 statelessd 来发布消息

要向 RabbitMQ 发布消息，任何标准的 HTTP 库都应该这样做。在这个例子中，我们将使用 Python 库 `requests`。在发布消息之前，首先应当创建并绑定队列用来发布消息。下列“7.4.4 Queue Setup”笔记文件中的代码实现了上述功能。

```
import rabbitpy
with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        queue = rabbitpy.Queue(channel, 'statelessd-messages')
        queue.declare()
        queue.bind('amq.topic', '#')
```

在声明队列后，剩下的就是发布消息。Statelessd 应该已经在 Vagrant 虚拟机中运行，因此从“7.4.4 节发布消息”运行以下代码将向“无状态消息”队列发布消息。

```
import requests
payload = {'body': 'from statelessd', 'app_id': 'example'}
response = requests.post('http://localhost:8900/%2f/amq.topic/example',
                         auth=('guest', 'guest'),
                         data=payload)
```

要验证消息是否已发布，请访问 http://localhost:15672/#/queues/%2F/statelessd-messages 的 RabbitMQ 管理界面。

现在你已经可以通过 statelessd 发布了消息。值得重申的是，statelessd 解决了将消息发布到 RabbitMQ 的特定用例场景。当考虑把 statelessd 集成到现有的消息架构中去时，别忘了 statelessd 的目标和性能。它旨在支持来自许多不同发布应用程序的高速发布。它不支持完整的 AMQP 协议，并且不支持 RabbitMQ 中许多更高级的消息发布功能，例如发布者确认或事务发布。值得一提的是，它确实让不少项目如虎添翼。

9.4　小结

通过支持 STOMP 和 MQTT 等替代协议，RabbitMQ 超越了作为 AMQP 厂商和平台中立的目标。此外，充满活力的插件和应用程序生态系统使得应用程序能以不同的方式与 RabbitMQ 进行消息通信。例如，对于容易出现网络中断和传输速度较慢的移动应用程序来说，不要使用像 AMQP 这样的协议，而应使用为这类任务专门设计的 MQTT 协议。像 Hare 和 statelessd 这样的应用程序正是为更有效的消息发布而生的。

除此以外，下列插件为 RabbitMQ 添加了替代协议的支持：

- rabbithub——为 RabbitMQ 添加 PubSubHubBub 支持 (https://github.com/ tonyg/ rabbithub)。
- udp_exchange——使用 UDP 协议向 RabbitMQ 发布消息 (https://github .com/tonyg/ udp-exchange)。
- rabbitmq-smtp——RabbitMQ 中 SMTP 转 AMQP 的网关 (https://github.com /rabbitmq/ rabbitmq-smtp)。
- rabbitmq-xmpp——RabbitMQ 中 XMPP 转 AMQP 的网关 (https://github .com/tonyg/ rabbitmq-xmpp)。

这些例子展示了 RabbitMQ 中各种各样的消息协议。尽管与 Web STOMP 插件相比，rabbitmq-smtp 插件的使用场景有所限制，但应用程序正好有这样独特的需求。我鼓励你在与 RabbitMQ 进行通信时确保选择正确的工具。

第 10 章　数据库集成

本章概要

- 从 PostgreSQL 发布 AMQP 消息
- 让 RabbitMQ 监听来自 PostgreSQL 的通知
- 使用 InfluxDB 存储交换器来存储消息

使用 RabbitMQ 可以解耦 OLTP 数据库的写操作。这是一种用来实现数据仓库和基于事件流处理的技术的常见方式。当发送序列化数据并写入数据库时，消费者应用程序在事件和数据库间扮演着桥梁的角色。其实消费者这一步也可以完全跳过。我们可以采用 RabbitMQ 插件，例如 InfluxDB 存储交换器。它可以自动将来自 RabbitMQ 的消息存储到数据库中。

RabbitMQ 和外部数据库之间的集成不止于此。另一种强大的应用模式是直接从数据库向 RabbitMQ 发送消息。我们可以通过使用数据库的扩展或者插件来实现上述功能。另一种方式是将 RabbitMQ 插件作为数据库的客户端，数据库一有变化时就发布消息。

注意 这两种数据库的集成模式可以简化运营复杂性，减少对外部消费者应用程序的需求。但这种简化操作也是有代价的。由于数据库和 RabbitMQ 之间耦合更加紧密，故障场景将变得更加复杂。举例来说，当 RabbitMQ 尝试向数据库服务器插入数据时，数据库服务器变慢或者干脆没有响应时该怎么办呢？在正式投入产品之前，通过对故障场景的针对性测试并明确故障排除和恢复步骤是很重要的。

本章将介绍与 RabbitMQ 的数据库集成模式。首先，你将学习如何安装 PostgreSQL 插件 pg_amqp，使用存储过程来发送消息。之后，你将学习如何使用 PostgreSQL 的 LISTEN 交换器的 `LISTEN/NOTIFY` 功能来实现同样的行为。在这之后，我们将离开关系数据库的世界，迈向 NoSQL 的世界。你将了解到如何利用 InfluxDB 存储交换器来存储发往 RabbitMQ 的时间序列数据。

10.1 PostgreSQL 扩展：pg_amqp

让 PostgreSQL 直接发布消息的想法并不新鲜。早在 2003 年，Slony 复制系统就采用 PostgreSQL 触发器功能来发送事件消息，实现了主从复制。在 2008 年，为了创建更为灵活的松耦合的复制系统来取代现存的复制系统，我创建了 Golcond。Golconde（https://code.google.com/p/golconde）采用 POST COMMIT 触发器和 PL/Python，采用 STOMP 消息通信协议，将事务数据发送到另一台 PostgreSQL 服务器。最新版 PostgreSQL 使用事件消息通信机制将事务数据发往用于热备份的另一个 PostgreSQL 实例。这个 PostgreSQL 实例扮演只读的从服务角色，并在主节点无响应时支持故障转移。

回顾这段 PostgreSQL 基于事件复制的往事，那么有人向生态系统中添加灵活的消息通信能力就不足为奇了。在 2009 年，来自 OmniTI 的 Theo Schlossnagle 发布了 pg_amqp。它是 PostgreSQL 的扩展，提供通过 PostgreSQL 函数发送 AMQP 消息的能力。虽然 pg_amqp 仅提供了 AMQP 0-8 规范的子集，但在从 PostgreSQL 触发器功能发送消息时表现优异。pg_amqp 提供的功能用起来就像其他 PostgreSQL 功能那样，而且可以通过 SQL 语句和存储过程来发起调用。pg_amqp 提供了两种方式与 RabbitMQ 交互：`amqp.publish` 和 `amqp.disconnect`。`amqp.publish` 方法将创建 AMQP 消息并使用 RPC 方法 `Basic.Publish` 将消息投递出去，就像任何其他 AMQP 发送方一样（见图 10.1）。连接将自动建立和销毁。如果想要在发送消息后直接终止连接的话，可以调用 `amqp.disconnect` 函数。

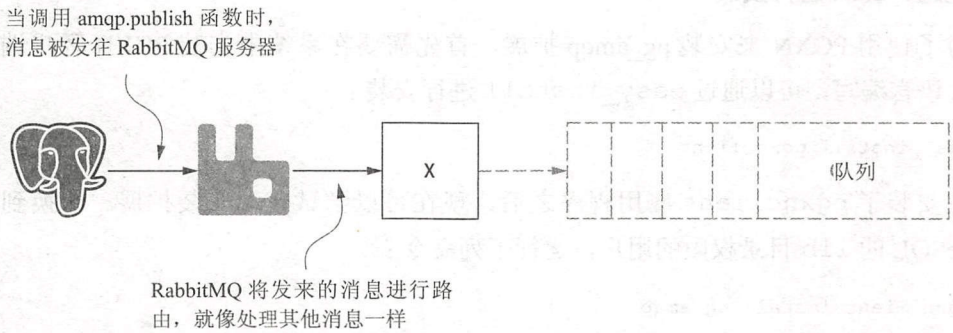


图 10.1　pg_amqp.publish 采用 Basic.Publish 将消息发往 RabbitMQ

在使用 pg_amqp 时，你发起的调用将与 RabbitMQ 进行同步通信，因此必须确保这一行为不会影响总体的查询速度。与任何紧密耦合的系统集成一样，我们应当在将其投入生产环境之前，执行基准测试并且测试失败场景。举例来说，如果你将 `amqp.publish` 方法调用包裹在事务中，而 pg_amqp 无法连接至 RabbitMQ 服务器时会发生什么呢？如果消息发送失败，数据库事务是否会完成呢？

让我们先安装 pg_amqp 扩展，再一探究竟。

10.1.1　安装 pg_amqp 扩展

有两种方法来安装 pg_amqp 扩展。可以直接下载并手工编译源代码。或者通过使用 PostgreSQL Extension Network(PGXN) 客户端。PGXN（https://pgxn.org）是 PostgreSQL 扩展的包管理仓库。基于 PGXN 的安装出奇的简单，只不过它无法兼容 PostgreSQL 9.3 安装。除非你使用的是 PostgreSQL 9.3 及后续版本，否则我建议你使用 PGXN 进行安装。假如安装失败的话再尝试手工安装。

注意　在尝试安装 pg_amqp 之前，应当先确保完全安装了 PostgreSQL 9.1 或之后的版本，包括开发文件。这是因为扩展程序是在安装过程中进行编译的。此外，你需要一系列工具从源代码编译 PostgreSQL。如果在安装 PostgreSQL 过程中遇到问题或者在使用开发工具编译 PostgreSQL 遇到困难，可以在官方 Wiki（https://wiki.postgresql.org/wiki/Detailed_installation _guides）上找到安装指引。

通过 PGXN 进行安装

为了使用 PGXN 来安装 pg_amqp 扩展，首先需要在系统中安装 PGXN 客户端。它由 Python 语言编写，可以通过 easy_install 进行安装：

```
easy_install pgxnclient
```

在安装了 pgxnclient 应用程序之后，现在可以尝试自动安装扩展。切换到有写入 PostgreSQL 的 lib 目录权限的用户，运行下列命令：

```
pgxnclient install pg_amqp
```

如果一切正常，该命令应当不会返回错误。如果遇到错误的话也不必担心，虽然手工安装需要更多步骤，但是也一样简单。

手工安装

你可以在 GitHub（https://github.com/omniti-labs/pg_amqp）上找到 pg_amqp 的源代码。如果你对 Git 不熟悉的话，可以从 https://github.com/omniti-labs/pg_amqp/archive/v0.3.0.zip 上下载源代码，并解压缩到目录中，然后进行编译。下列清单中所展示的代码是用 BASH 编写的。它修复了 PostgreSQL 9.3 及以后系统的安装，你需要到解压代码的顶层目录中运行。

清单 10.1　编译并安装 pg_amqp

```
#!/bin/bash
LIBDIR=`pg_config --libdir`
INSTALLSH="$LIBDIR/pgxs/config/install-sh"
make && make INSTALL=$INSTALLSH install
```

在成功安装 pg_amqp 扩展之后，你需要将其加载到 PostgreSQL 数据库中。为了便于演示，下列示例代码使用默认的 postgres 超级用户和 postgres 数据库。当然，这并不是必须的，但是用户必须是超级用户。

为了加载扩展，需要使用 psql 连接到 PostgreSQL 数据库：

```
$ psql -U postgres postgres
```

连接上之后，你将看到类似下列内容的输出：

```
psql (9.3.5)
Type "help" for help.
postgres=>
```

现在，你可以使用 CREATE EXTENSION 语法来加载扩展：

```
postgres=> CREATE EXTENSION amqp;
```

如果扩展加载成功的话，你将收到如下类似的信息：

```
CREATE EXTENSION
```

在加载完扩展之后，你就可以开始对扩展进行配置，然后发送消息。

10.1.2　配置 pg_amqp 扩展

上一节中运行 CREATE EXTENSION 命令时，表 amqp.broker 就已自动创建了。我们现在要做的就是填写该表来配置扩展。如表 10.1 所示，amqp.broker 包含了普通连接设置，其中 broker_id 字段在调用 amqp.publish 和 amqp.disconnect 函数时会用到。

表 10.1　amqp.broker 表定义

列	类　型	修　饰　符
broker_id	Integer	not null default nextval('broker_broker_id_seq')
host	Text	not null
port	Integer	not null default 5672
vhost	Text	
username	Text	
password	Text	

假设 PostgresSQL 在本地运行，同时 RabbitMQ 像前面的章节中那样运行在 Vagrant VM 中的话，那么你可以通过 localhost 进行连接。下列 SQL 语句将用来配置 pg_amqp。它会向表中插入一行，其中 localhost 表示连接到 RabbitMQ 的地址，端口为 5672，虚拟主机为 /，并且用户名和密码为 guest/guest 的组合。如果测试环境的不同导致连接配置无法正常工作的话，请调整 SQL 中对应的参数。

```
INSERT INTO amqp.broker (host, port, vhost, username, password)
  VALUES ('localhost', 5672, '/', 'guest', 'guest')
  RETURNING broker_id;
```

当执行该命令时，你将获取 broker_id，这意味着插入表格已经成功：

```
broker_id
-----------
         1
(1 row)
INSERT 0 1
```

记住这个 broker_id 值，因为你将用它来向 RabbitMQ 发送消息。

10.1.3 通过 pg_amqp 发送消息

在安装并配置完 pg_amqp 后，现在就可以发送第一条消息了。在这之前，需要在 RabbitMQ 管理界面配置队列以便接收消息。打开浏览器，访问 http://localhost:15672/#/queues，创建一个名为 pg_amqp-test 的队列，如图 10.2 所示。

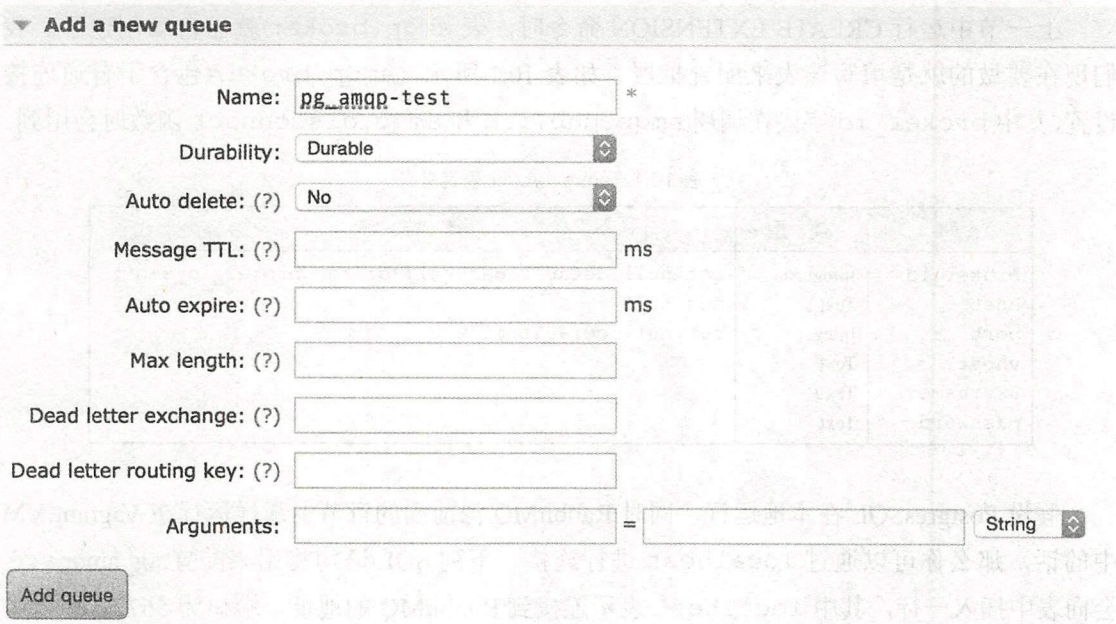


图 10.2 创建 pg_amqp-test 队列

当创建完队列后，你就可以通过默认的 direct 交换器，使用队列名称 pg_amqp-test 作为路由键，将消息从 PostgreSQL 发往 RabbitMQ 队列。使用 psql 链接到 postgres 数据库，发送下列查询命令，参数包括服务器 ID、交换器、路由键和消息内容：

```
SELECT amqp.publish(1, '', 'pg_amqp-test',
                    'Test message from PostgreSQL');
```

提交之后，你应当收到查询执行成功的确认结果：

```
publish
---------
t
(1 row)
```

虽然 PostgreSQL 提示消息已被发送出去，但更好的验证方式是使用管理界面来获取发送出去的消息。在队列详情页 http://localhost:15672/#/queues/%2F/pg_amqp-test 上，你可以通过 Get Messages 模块来获取和查看消息。如图 10.3 所示，一旦单击了 Get Message(s) 按钮，你应当能看到通过 PostgreSQL 的 `amqp.publish` 函数发送出去的消息。

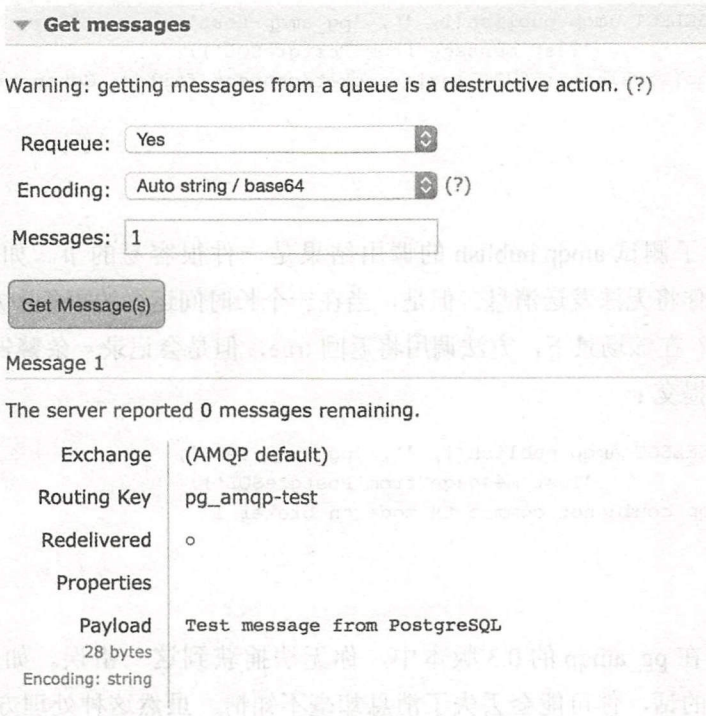


图 10.3　使用管理界面来确认消息已经被发送出去

在完成了设置和配置之后，通过 pg_amqp 发送消息就显得轻而易举了。值得注意的是，你无法设置 0.3 版本的 AMQP 消息属性。此外，消息发送是包装在一个 AMQP 事务当中的。你应当在 PostgreSQL 事务中调用 `amqp.publis` 函数。如果事务回滚的话，RabbitMQ 事务也会回滚。大多数情况下，消息发送的代码会包装在一段存储过程中。要么在存储过程中执行其他操作，要么作为表中行的 INSERT、UPDATE 和 DELETE 的触发器函数。

注意　管理界面会警告你 Get Messages(s) 操作是一种破坏性操作。这是因为消息将从队列中移除后再进行展示。同时，如果 Requeue 选项被设置为 Yes 的话，RabbitMQ 将会把消息重新发送回队列，这时消息将在队列的最底端。

处理故障

你可能注意到了在调用 amqp.publish 函数时，它会返回一个布尔值。在成功的场景下，它将返回一个 t，代表 true。但如果 PostgreSQL 无法连接 RabbitMQ 该怎么办呢？我们新起一个事务，在新建立的连接中发送相同的语句将会得到 f(false) 的响应，并打印出一条警告日志：

```
postgres=# SELECT amqp.publish(1, '', 'pg_amqp-test',
                    'Test message from PostgreSQL');
WARNING: amqp[localhost:5672] login socket/connect failed: Connection refused
publish
---------
f
(1 row)
```

该场景展示了测试 amqp.publish 的调用结果是一件很容易的事。如果结果返回的是 false 的话，那么你将无法发送消息。但是，当在一个长时间运行的事务中发生 RabbitMQ 断连的话会怎样呢？在该场景下，方法调用将返回 true，但是会记录一条警告日志，内容就是 AMQP 事务无法提交：

```
postgres=# SELECT amqp.publish(1, '', 'pg_amqp-test',
                    'Test message from PostgreSQL');
WARNING: amqp could not commit tx mode on broker 1
publish
---------t
(1 row)
```

不幸的是，在 pg_amqp 的 0.3 版本中，你无法捕获到这一错误。如果没有时刻监视 PostgreSQL 日志的话，你可能会丢失了消息却毫不知情。虽然这种处理方式欠妥当，但是总比丢失数据库事务要好些。就像所有运营系统一样，监控才是解决问题的关键。如果你使用类似 Splunk 的系统，你可以创建一个任务来周期性地在 PostgreSQL 日志中搜索 AMQP 错误。另一种方案是自行编写应用或者插件来实现类似 Nagios 一样的功能，在日志中搜索特定的警告信息。

10.2 监听 PostgreSQL 通知

虽然 pg_amqp 提供了一种方便快速地直接从 PostgreSQL 发送消息的方式，但同时它也在服务器实例之间建立了紧密的耦合关系。一旦 RabbitMQ 集群由于某种原因变得不可用的话，那么这将对 PostgreSQL 服务器造成不利的影响。为了避免紧耦合同时保留直接集成的方式，我创建了 PostgreSQL LISTEN 交换器。

PostgreSQL LISTEN 交换器的作用就像 PostgreSQL 客户端一样，监听由 NOTIFY SQL 语句在 PostgreSQL 中发出的通知。PostgreSQL 通知将被发往一条信道，即客户端订阅的文本值。该值在 PostgreSQL 的 LISTEN 交换器中被用作路由键。当通知消息被发往 LISTEN 交换器所注册的信道上时，那么该通知将会被转换成一条消息，然后发往类似 direct 类型的交换器上（见图 10.4）。

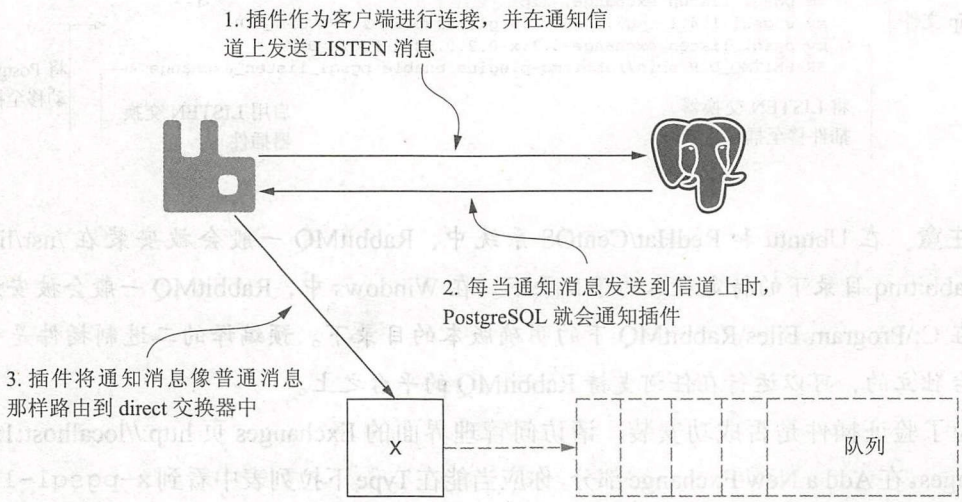


图 10.4　LISTEN 交换器功能类似 PostgreSQL 客户端，将通知消息像普通消息那样发送出去

当然，所有技术决策都需要权衡利弊。有了 pg_amqp 之后，如果无法连接至 RabbitMQ 的话，对 amqp.publish 的调用就会失败。而使用 LISTEN 交换器时，一旦 PostgreSQL 连接失效后，就不能再注册消息通知了，也就是说不能再发送任何消息了。你可以使用 RabbitMQ 管理 API 来监控交换器吞吐量的方式来鉴别这一场景。

10.2.1　安装 PostgreSQL LISTEN 交换器

PostgreSQL LISTEN 交换器可以从 GitHub（https://github.com/AWeber/pgsql-listen-exchange）上进行下载。在项目页面展示的 README 文件中，包含特定 RabbitMQ 版本的预编译二进制插件的下载。在下载和安装插件前，请确保下载了对应 RabbitMQ 版本的最新插件版本。

下载的压缩文件中包含了两个 RabbitMQ 插件：一个交换器和 PostgreSQL 驱动。下列代码清单将在 OS X 系统上下载并安装插件。该 OS X 系统运行着通过 Homebrew 安装的 RabbitMQ 3.3.5。对于其他系统来说，你需要修改 `RABBITMQ_DIR` 的赋值，将其指向 RabbitMQ 的基本目录。

清单 10.2 LISTEN 交换器的 OS X 安装脚本

```
#!/bin/bash
RABBITMQ_DIR=/usr/local/Cellar/rabbitmq/3.3.5/
PLUGIN_DIR=$RABBITMQ_DIR/plugins/
cd /tmp
curl -L -o pgsql-listen-exchange.zip http://bit.ly/1ndl8eK
unzip pgsql-listen-exchange.zip
rm pgsql-listen-exchange. zip
mv epgsql-1.4.1-rmq3.3.x-0.2.0-git3318bd5.ez $PLUGIN_DIR
mv pgsql_listen_exchange-3.3.x-0.2.0.ez $PLUGIN_DIR
$RABBITMQ_DIR/sbin/rabbitmq-plugins enable pgsql_listen_exchange
```

设置 RabbitMQ 安装的基本目录
设置插件目录路径
从 GitHub 上下载交换器
变更到临时目录
移除下载的 zip 文件
解压 zip 文件
将 PostgreSQL 驱动移至插件目录
将 LISTEN 交换器插件移至插件目录
启用 LISTEN 交换器插件

注意 在 Ubuntu 和 RedHat/CentOS 系统中，RabbitMQ 一般会被安装在 /usr/lib/rabbitmq 目录下的特定版本的子目录下。在 Windows 中，RabbitMQ 一般会被安装在 C:\Program Files\RabbitMQ 下的明确版本的目录下。预编译的二进制插件是平台独立的，可以运行在任何支持 RabbitMQ 的平台之上。

为了验证插件是否成功安装，请访问管理界面的 Exchanges 页 http://localhost:15672/#/exchanges。在 Add a New Exchange 部分，你应当能在 Type 下拉列表中看到 `x-pgsql-listen` 值（见图 10.5）。

在验证成功安装之后，现在可以对交换器进行配置了。如果在下拉列表中没有找到这一选项的话，要么就是插件没有复制到正确的目录中，要么可能就是你运行 Erlang 的版本要比插件所编译的版本要早。强烈建议使用 Erlang R16 及以后的版本。

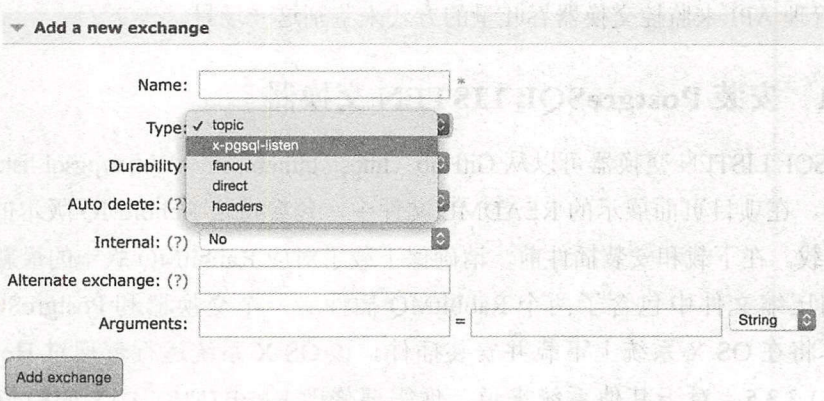


图 10.5 验证在 Type 下拉列表中确实存在 x-pgsql-listen 这一选项

有多种方式来配置插件。可以在声明交换器时传入连接参数的方式来直接配置插件，也可以在 rabbitmq.config 文件中配置插件，还可以通过应用在交换器上的策略来配置插件。策略提供了最为灵活的交换器配置方式。你应当使用该方式直到对插件的使用了如指掌。

10.2.2　基于策略的配置

访问管理界面的 Amin 页。在该页的右侧可以看到 Policies（见图 10.6）。

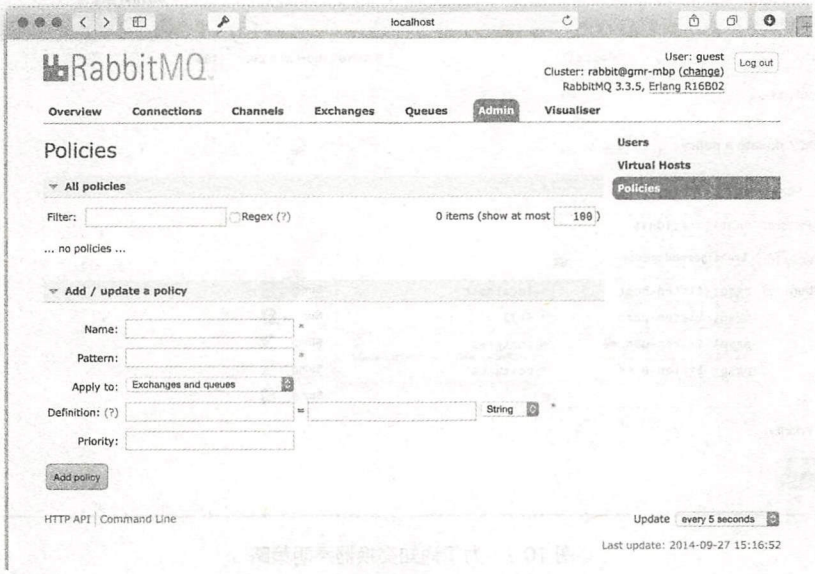


图 10.6　管理界面中的 Policies 页

为了创建策略以便连接至 PostgreSQL，需要指定策略名称，用来匹配交换器名称的正则表达式，以及 PostgreSQL 主机、端口、数据库名称、用户名和可选的密码。此外，你

可以指定将其仅应用于交换器来缩小策略的应用范围。如图 10.7 所展示的策略，将连接至 localhost、端口为 5432 的 PostgreSQL，使用 postgres 作为数据库名和用户名。

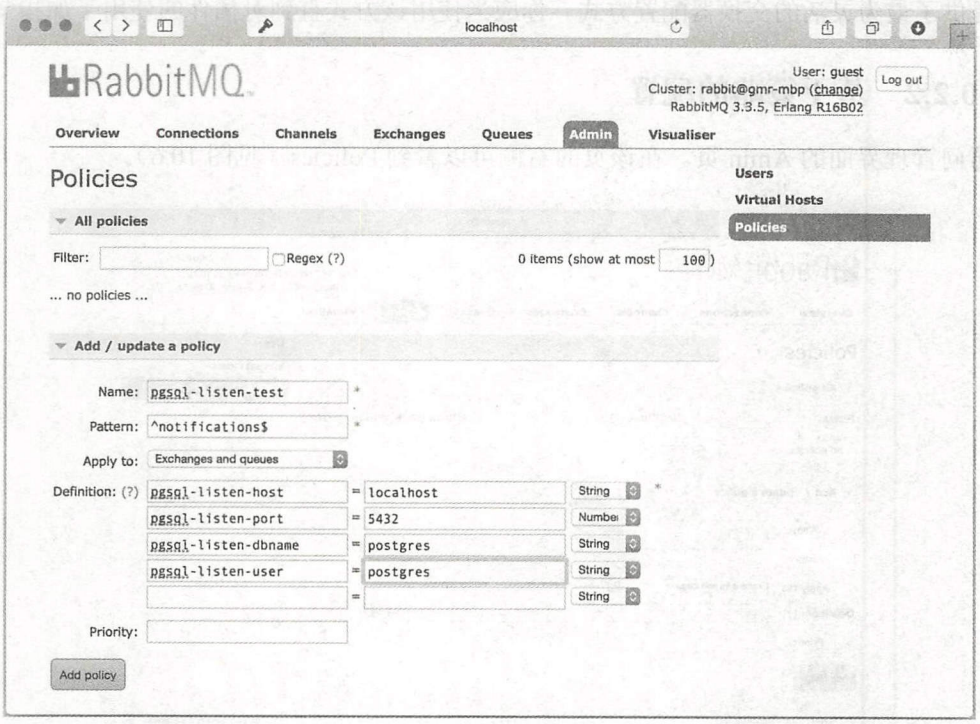


图 10.7　为了通知交换器声明策略

单击 Add Policy 后，你将在当前页看到如图 10.8 所示的策略。在添加策略时，你提供的连接信息将会被检测类型正确与否，而不会检测其有效性。直到交换器创建时，才会检测连接信息的有效性。

名　称	模　式	应 用 于	定　义	优 先 级
pgsql-listen-test	^notifications$	exchanges	pgsql-listen-host:localhost pgsql-listen-port:5432 pgsql-listen-dbname:postgres pgsql-listen-user:postgres	0

图 10.8　添加至 All Policies 模块的策略

10.2.3　创建交换器

在创建完策略之后，访问管理界面的 Exchanges 页（http://localhost:15672/#/exchanges）。利用页面底部区域的 Add a New Exchang 表单创建一个新的交换器。将交换器命名为 `notification`，以便策略能够匹配上，同时将交换器类型设置为 `x-pgsql-listen`（见图 10.9）。

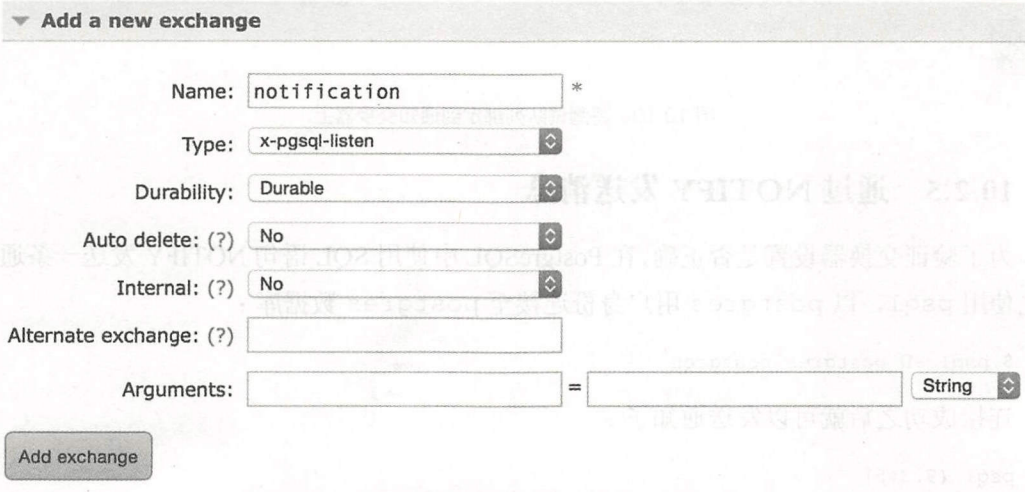


图 10.9　添加用于通知的 PostgreSQL LISTEN 交换器

交换器添加完成之后，它会连接到 PostgreSQL，但不会监听通知。为了让它能启动监听交换器通知，必须绑定到交换器，使用匹配 PostgreSQL 通知信道字符串作为路由键。

10.2.4　创建并绑定测试队列

测试 LISTEN 交换器配置的最后一步是创建一个测试队列，并把通知消息发送给它。访问管理界面的 Queues 页（http://localhost:15672/#/queues），在 Add a New Queue 模块中创建队列。为了测试目的，需要将队列命名为 `notification-test`。在添加队列时，无需改变表单中的其他任何自定义参数和默认配置。

添加完队列之后，访问管理界面的队列页面（http://localhost:15672/#/queues/%2F/notification-test）。在页面上的 Bindings 模块中，创建绑定到 `notification` 交换器，并将路由键设置为 `example`（见图 10.10）。

添加完成之后，交换器将连接至 PostgreSQL，并执行 LISTEN 语句，注册所有通过 `example` 信道发送的通知。现在准备发送一条测试通知。

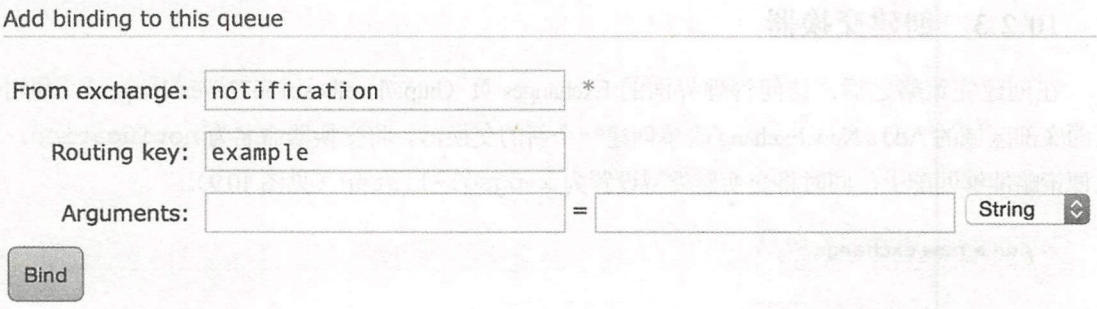


图 10.10　将测试队列绑定到通知交换器上

10.2.5　通过 NOTIFY 发送消息

为了验证交换器设置是否正确，在 PostgreSQL 中使用 SQL 语句 NOTIFY 发送一条通知。首先使用 psql，以 postgres 用户身份连接至 postgres 数据库：

```
$ psql -U postgres postgres
```

连接成功之后就可以发送通知了：

```
psql (9.3.5)
Type "help" for help.
postgres=# NOTIFY example, 'This is a test from PostgreSQL';
NOTIFY
```

在发送通知之后，切回 RabbitMQ 管理界面，在 Get Messages 模块里获取 notification-test 队列上的消息（见图 10.11）。

可以看到，LISTEN 交换器会给消息添加元数据信息。而在使用 pg_amqp 时并没有填充这些数据。消息属性中的 app_id 表明这条消息是由 pgsql-listen-exchange 插件产生的。另一个属性 timestamp 则取自 RabbitMQ 服务器的当前本地时间。另外设置了一些 headers，以表明 PostgreSQL 通知通道、数据库、服务器和源交换器的名字。

虽然本示例程序中采用的是纯文本，但也可以将数据序列化为不同的格式来发送通知，使得应用程序的通知功能变得丰富多彩。你可以采用这种方案来实现复杂存储过程的调试，这让跟踪数据库中的数据状态变得简单。你也可以用这种方式来更新云端的其他系统。这需要交换器和 RabbitMQ 联合插件的组合使用。不管是哪种场景，LISTEN 交换器使得系统与 PostgreSQL 能够以松耦合的方式进行集成，同时也仅带来的影响也不大。

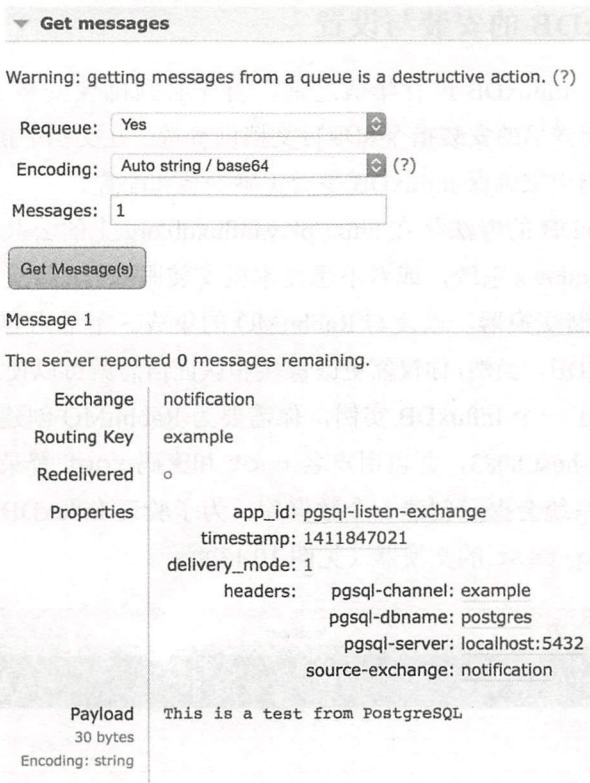


图 10.11　从 notification-test 队列获取消息

10.3　将消息存入 InfluxDB 中

InfluxDB (http://influxdb.com) 是采用 Go 语言实现的开源、分布式、基于时间序列的数据库。可以很方便地在 Linux 和 OS X 系统上进行配置。由于 InfluxDB 提供了多种易用的协议用来填充数据以及一个内建的、基于 Web 的查询界面用来查询存储的数据，这使得 InfluxDB 作为以分析为目的的时间序列数据存储系统格外引人注目。InfluxDB 很快成为诸如 Graphite 等这类系统的替代方案。这离不开它的高内聚、可扩展集群存储方案。

路由至 InfluxDB 存储交换器的消息会被检测是否应该存储在 InfluxDB 中。如果消息设置了内容类型，并且被设置为 application/json 的话，那么该消息就会被转换成合适的格式，并以路由键作为 InfluxDB 的时间名称存储于 InfluxDB 中。此外，如果设置了 timestamp 的话，那么就会和 InfluxDB 的事件 time 列自动映射。

10.3.1 InfluxDB 的安装与设置

在将 RabbitMQ 和 InfluxDB 进行集成之前，首先必须确保安装了 InfluxDB。在 http://influxdb.com/docs/ 上有详尽的安装指令和项目文档供参考。在文档中挑选最新的版本，然后按照安装指引和新手指引来确保 InfluxDB 是否正确安装和配置。

另一种学习 InfluxDB 的方法是在 http://play.influxdb.org 上的公共试验田服务器上做试验。如果使用的是 Windows 系统，或者不想在本机安装服务器的话，你就可以采用这种方法来测试 InfluxDB 存储交换器，以及与 RabbitMQ 的集成。本节内容中的示例程序会假设你在本地有安装 InfluxDB。当然，你仅需更改链接和认证信息就可以使用公共试验田服务器。

假设在本地配置了一个 InfluxDB 实例，你需要为 RabbitMQ 创建数据库和用户。在浏览器中打开 http://local-host:8083，并以用户名 `root` 和密码 `root` 登录进入管理界面。

在首次登录时，系统会提示创建一个数据库。为了验证 InfluxDB 存储交换器，我们创建一个名为 `rabbitmq-test` 的交换器（见图 10.13）。

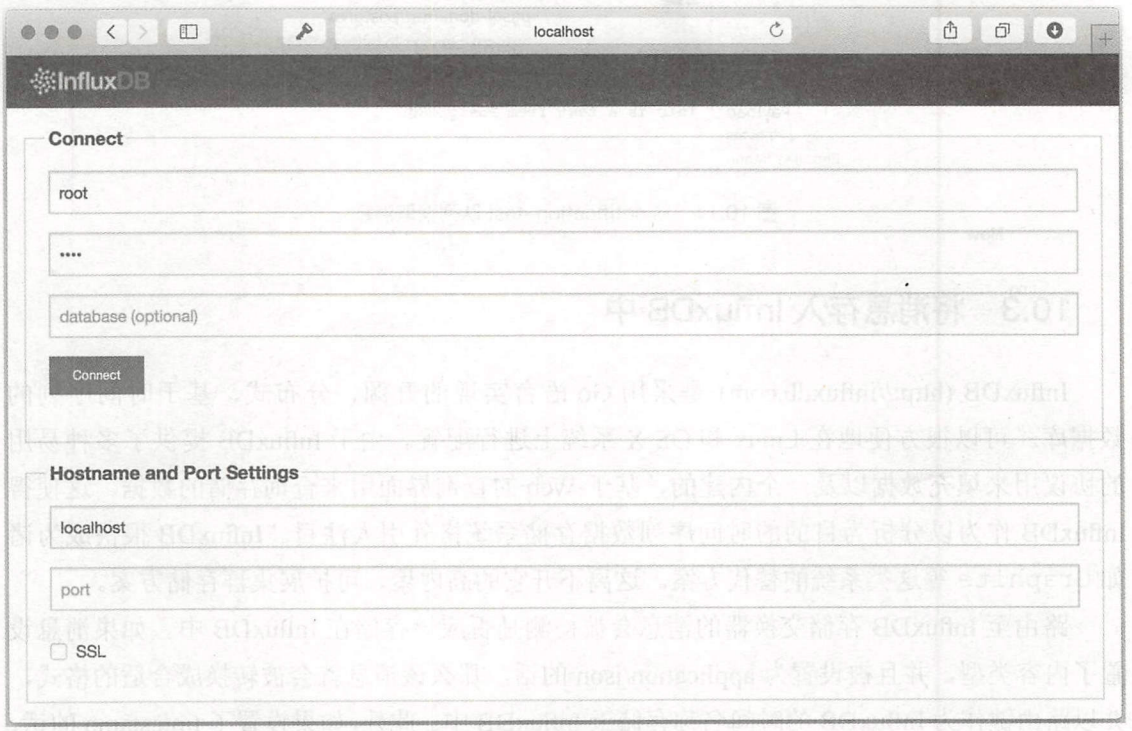


图 10.12　登录 InfluxDB 的管理界面

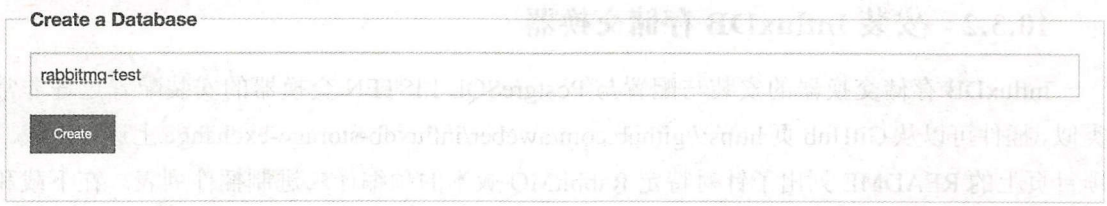


图 10.13　创建 rabbitmq-test 数据库

在数据库创建完成之后，它将展示在 Web 页的顶端列表中。单击列表中的 `rabbitmq-test` 进入数据库用户设置界面。在该页面上为 RabbitMQ 添加一位用户。插件将会使用该用户来认证 RabbitMQ（见图 10.14）。在表单中输入 `rabbitmq` 作为用户名，`test` 作为密码，并单击 Create 按钮。

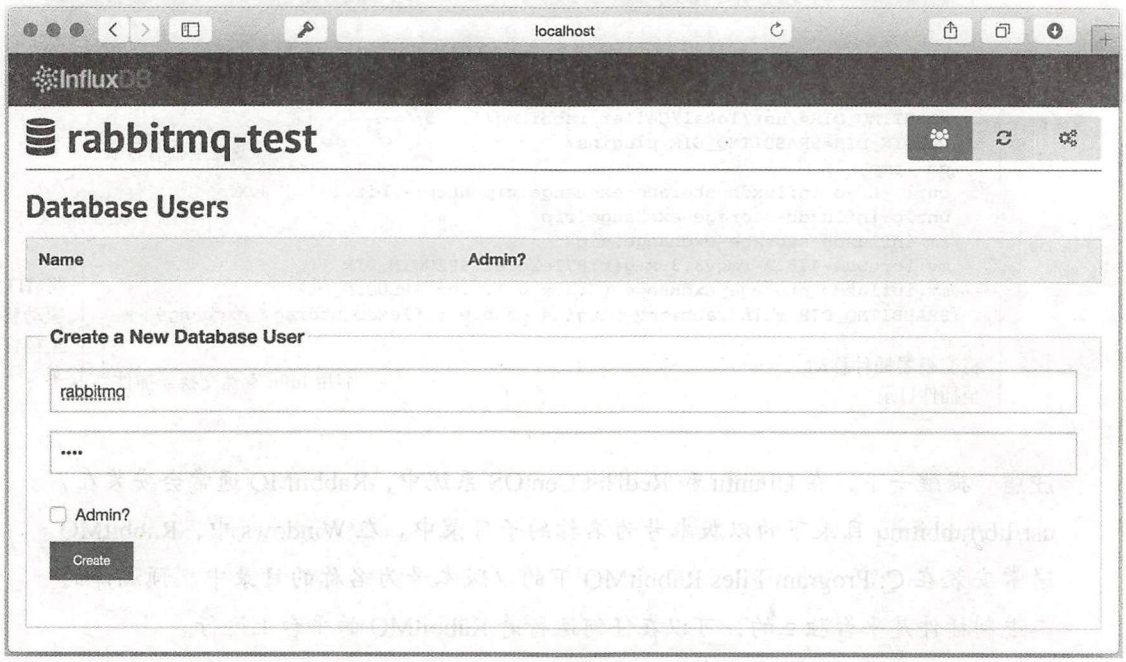


图 10.14　为 rabbitmq-test 数据库创建 rabbitmq 用户

在创建完用户之后，它将会展现在页面顶端的 Database Users 表格中。之后，你就可以安装并配置 InfluxDB 存储交换器了。

10.3.2 安装 InfluxDB 存储交换器

InfluxDB 存储交换器的安装与配置与 PostgreSQL LISTEN 交换器的安装配置过程非常类似。插件可以从 GitHub 页 https://github.com/aweber/influxdb-storage-exchange 上进行下载。项目页上的 README 列出了针对特定 RabbitMQ 版本的预编译二进制插件列表。在下载和安装插件时，请务必确保下载的是 RabbitMQ 版本对应的最新插件版本。

下载的压缩文件中包含了两个 RabbitMQ 插件：交换器和 HTTP 客户端库。假设在 OS X 系统中通过 Homebrew 安装了 RabbitMQ 的 3.3.5 版本，那么下列代码清单将在此基础之上下载并安装插件。对于其他系统来说，你需要修改 RABBITMQ_DIR 变量的赋值，将其指向正确的 RabbitMQ 基本目录。

清单 10.3 InfluxDB 存储交换器的 OS X 安装脚本

```
#!/bin/bash
RABBITMQ_DIR=/usr/local/Cellar/rabbitmq/3.3.5/
PLUGIN_DIR=$RABBITMQ_DIR/plugins/
cd /tmp
curl -L -o influxdb-storage-exchange.zip http://bit.ly/1j7UvXf
unzip influxdb-storage-exchange.zip
rm influxdb-storage-exchange.zip
mv ibrowse-4.0.2-rmqv3.3.x-git7871e2e.ez $PLUGIN_DIR
mv influxdb_storage_exchange-v3.3.x-0.1.1.ez $PLUGIN_DIR
$RABBITMQ_DIR/sbin/rabbitmq-plugins enable influxdb_storage_exchange
```

设置 RabbitMQ 的基本目录

设置插件目录

从 GitHub 上下载交换器

变更至临时目录

移除下载的 zip 文件

解压 zip 文件

将 HTTP 驱动移动至插件目录

将交换器插件移动至插件目录

启用 Influ 存储交换器插件

注意 提醒一下，在 Ubuntu 和 RedHat/CentOS 系统中，RabbitMQ 通常会安装在 /usr/lib/rabbitmq 目录下的以版本号为名称的子目录中。在 Windows 中，RabbitMQ 通常安装在 C:\Program Files\RabbitMQ 下的以版本号为名称的目录中。预编译的二进制插件是平台独立的，可以在任何运行着 RabbitMQ 的平台上运行。

为了验证插件安装正确与否，打开 http://localhost:15672/#/exchanges 进入管理界面的 Exchanges 页。在 Add a New Exchange 模块中，你应当能在 Type 下拉列表中看到 `x-influxdb-storage` 值（见图 10.15）。

确认无误之后，现在就可以创建 InfluxDB 存储交换器实例了。

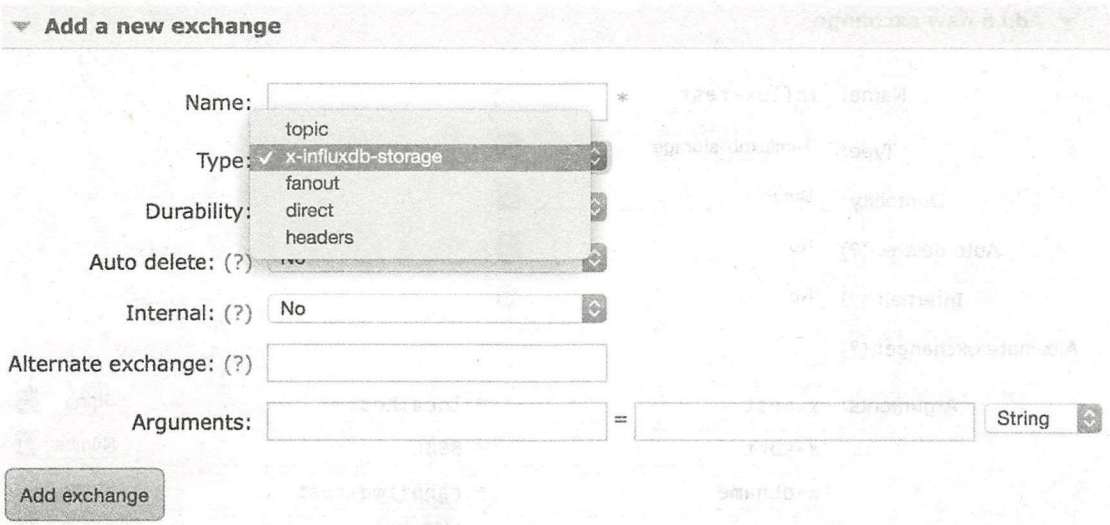


图 10.15　验证 InfluxDB 存储交换器正确安装

10.3.3　创建测试交换器

类似于 PostgreSQL LISTEN 交换器，InfluxDB 存储交换器可以通过策略或者 rabbitmq.config 进行配置，也可以在声明交换器的时候传入自定义参数进行配置。关于配置选项以及配置方法参数的说明与用法，请参考 GitHub（https://github.com/aweber/ influxdb-storage-exchange）上的 README 文档。为了展示 PostgreSQL LISTEN 交换器中基于策略的配置方式与基于参数的配置方式两者之间的差异，在接下来的例子中，我们将使用基于参数的配置方式来创建交换器。

首先，访问 http://localhost:15672/#/exchanges，单击 RabbitMQ 管理界面上的 Exchanges 页，进入 Add a New Exchang 模块。自定义参数的交换器配置方法是通过前缀参数 x- 完成的。这些以 x- 开头的参数不是标准的 AMQP 或者 RabbitMQ 参数。你需要配置 InfluxDB 连接的主机地址、端口、数据库名称、用户名和密码。这些值都以 x- 作为前缀，如图 10.16 所示。如果不将这些参数以 x- 打头的话，交换器将会采用默认值。

在添加交换器时，这些参数将会被验证类型是否合法，但不会测试连接信息是否正确。由于AMQP交换器的不可变性，如果错误配置了交换器的话，那么你需要将其删除并重新添加。

发往交换器的消息首先会被存储到 InfluxDB 中，然后被路由到以 topic 交换器路由键方式绑定至交换器的队列或者交换器上。错误配置的交换器不会阻碍消息的路由，但是会使得消息无法存储到 InfluxDB 中。

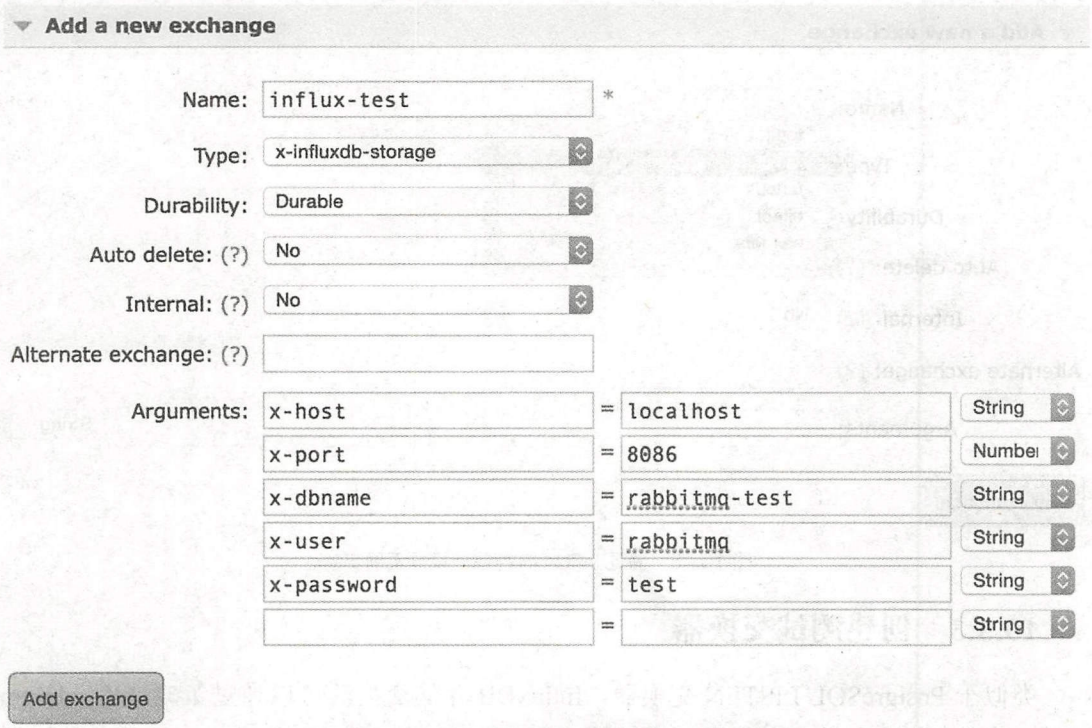


图 10.16　基于参数配置的方式添加 InfluxDB 交换器

在创建完交换器后，现在就可以向其发送消息来进行测试了。如果采用 rabbitmq.config 或者基于策略的配置方式的话，那么可以将这些参数留空。同时，方法的值将会被应用到交换器的配置上。

10.3.4　测试交换器

为了测试集成得正确与否，访问 http://localhost:15672/#/exchanges/%2F/ influx-test，进入 RabbitMQ 管理界面的新建交换器页面。在 Publish a Message 模块中，将消息的 content_type 设置为 `application/json`，设置有效的 `timestamp` 值，以及正确的 JSON 消息体（见图 10.17）。

由于没有将队列绑定至交换器，在发送消息时你将收到一条消息没有被路由的警告。这没有什么关系，因为你只是想确认数据是否已经存入 InfluxDB 中。

为了确认消息已经正确存储，打开浏览器，访问 http://localhost:8083 所在的管理界面，然后以 root 用户（密码 root）登录。之后你将能看到如图 10.18 所示的数据库列表。单击 `rabbitmq-test` 数据库的 Explore Data 链接。

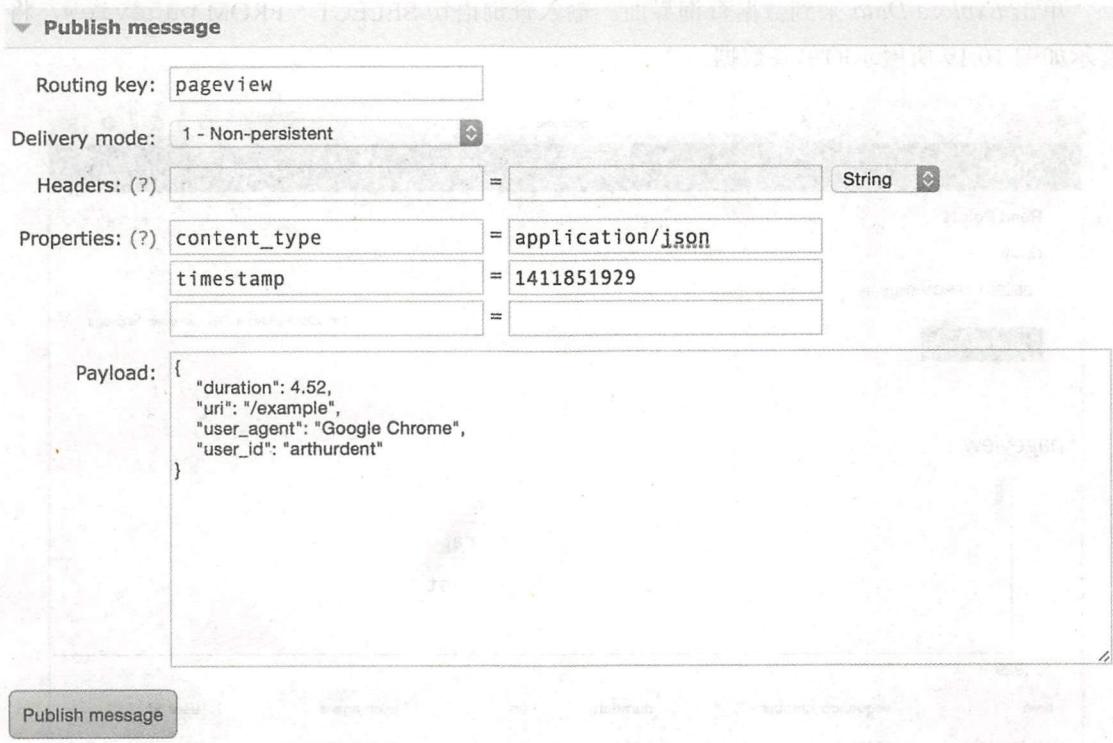


图 10.17　将 JSO 消息发送至 InfluxDB 存储交换器

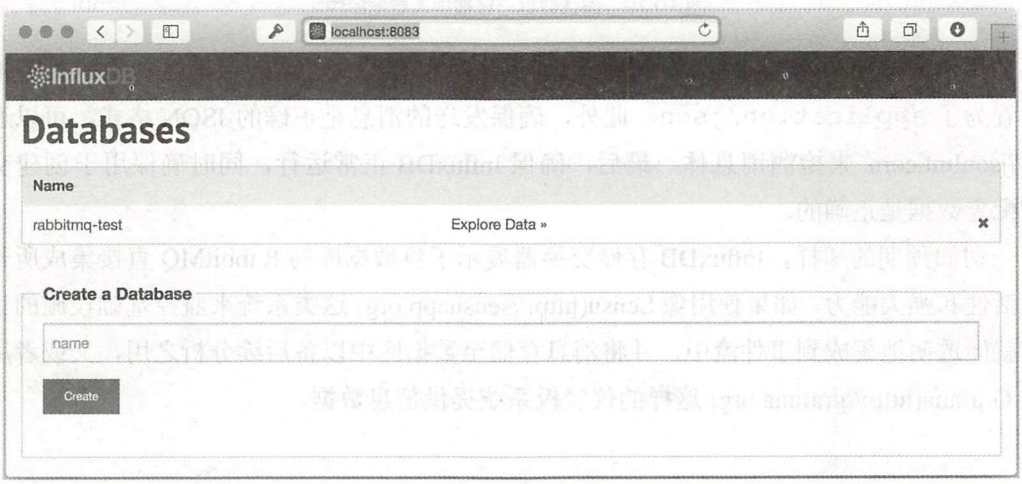


图 10.18　InfluxDB 管理界面上展示了数据库列表

单击 *Explore Data* 来到数据查询界面。输入查询语句 SELECT * FROM `pageview`，将展示如图 10.19 所展示的单条数据。

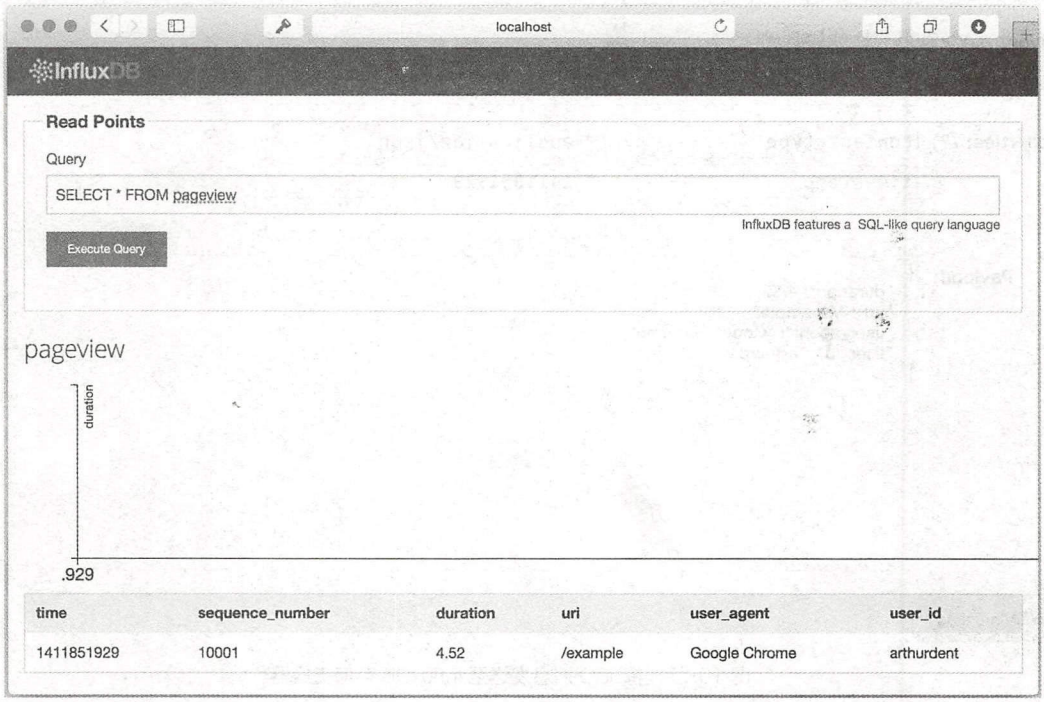


图 10.19　确认数据已经被插入数据库中

如果没有显示查询结果的话，也许消息头信息中有错误。请确保消息的内容类型被正确设置为了 `application/json`。此外，确保发送的消息是正确的 JSON 格式。可以通过 http://jsonlint.com. 来检测消息体。最后，确保 InfluxDB 正常运行，同时确保用于创建交换器的配置数据是正确的。

一切如预期的那样，InfiuxDB 存储交换器展示了将数据库与 RabbitMQ 直接集成所带来的灵活性和强大能力。如果使用像 Sensu(http://sensuapp.org) 这类系统来监控基础设施的话，那么就能透明地集成到事件流中，并将消息存储至数据库中以备后续分析之用，又或者用来为像 Grafana(http://grafana.org) 这样的仪表板系统提供信息数据。

10.4　小结

将数据库与 RabbitMQ 进行集成，减少了在数据库或者 RabbitMQ 栈外运行消费者或消息发送应用程序的运营成本。不过这种简化也是有成本的。由于 RabbitMQ 和数据库之间紧密耦合在了一起，故障场景就变得更为复杂了。

在本章中你学习了如何将 PostgreSQL 用作消息的来源，将消息通过 PostgreSQL 扩展 pg_amqp 或者 PostgreSQL LISTEN 交换器路由到 RabbitMQ。同时，本章还详述了安装和使用 InfluxDB 存储交换器，演示了发送至 RabbitMQ 的消息时如何通过 RabbitMQ 存储至数据库中的。

本章中介绍的数据库集成仅仅是冰山一角。还有许多项目可以将数据库直接和 RabbitMQ 集成起来。例如 Riak 交换器 (https://github.com/jbrisbin/riak-exchange) 及其对应的项目。它实现了 Riak RabbitMQ 提交钩子（commit hooks）(https://github.com/jbrisbin/riak-rabbitmq-commit-hooks)，当 Riak 中发生写事务时会发送消息至 RabbitMQ。想要了解有哪些数据库插件可供选择的话，可以访问 RabbitMQ Community Plugins（https://www.rabbitmq.com/community-plugins.html ）和 RabbitMQ Clients & Developer Tools（https://www.rabbitmq.com/devtools.html）。

没有找到需要的插件？期待你能贡献另一个将数据库与 RabbitMQ 集成的插件！

附录

准备就绪

本附录介绍了如何安装配置 VirtualBox 和 Vagrant。本书采用的 Vagrant 虚拟机（VM）包含了所有需要用来测试代码的样例、示例程序，并与本书的内容配套。

我们将采用 VirtualBox 这款虚拟化软件来运行所有的示例程序。而 Vagrant 则是用来配置虚拟机的自动化工具。首先，你需要安装这两个应用程序，然后下载包含 Vagrant 配置和 Chef 参考手册的 zip 文件，用来配置本书对应的 VM。下载完成并解压后，你仅需运行一条命令就能启动虚拟机，同时你能交互式地测试本书中的代码清单和示例。

不论是在 Windows、OS X 还是在 Linux 中，安装过程都非常简单。这是因为它旨在让配置步骤尽可能地精简。首先，你需要下载并安装 VirtualBox。

A.1 安装 VirtualBox

VirtualBox 是款免费的虚拟化产品，最初由 Sun 微系统公司开发，现在由 Oracle 提供支持。它运行在 Windows、Linux、Macintosh 和 Solaris 系统上，并提供用于本书的 VM。

VirtualBox 的配置简单明了。你可以从 http://virtualbox.org 上进行下载。只需打开下载页面，选择适合于你使用的操作系统类型的 VirtualBox 平台包，并下载安装包即可（见图 A.1）。

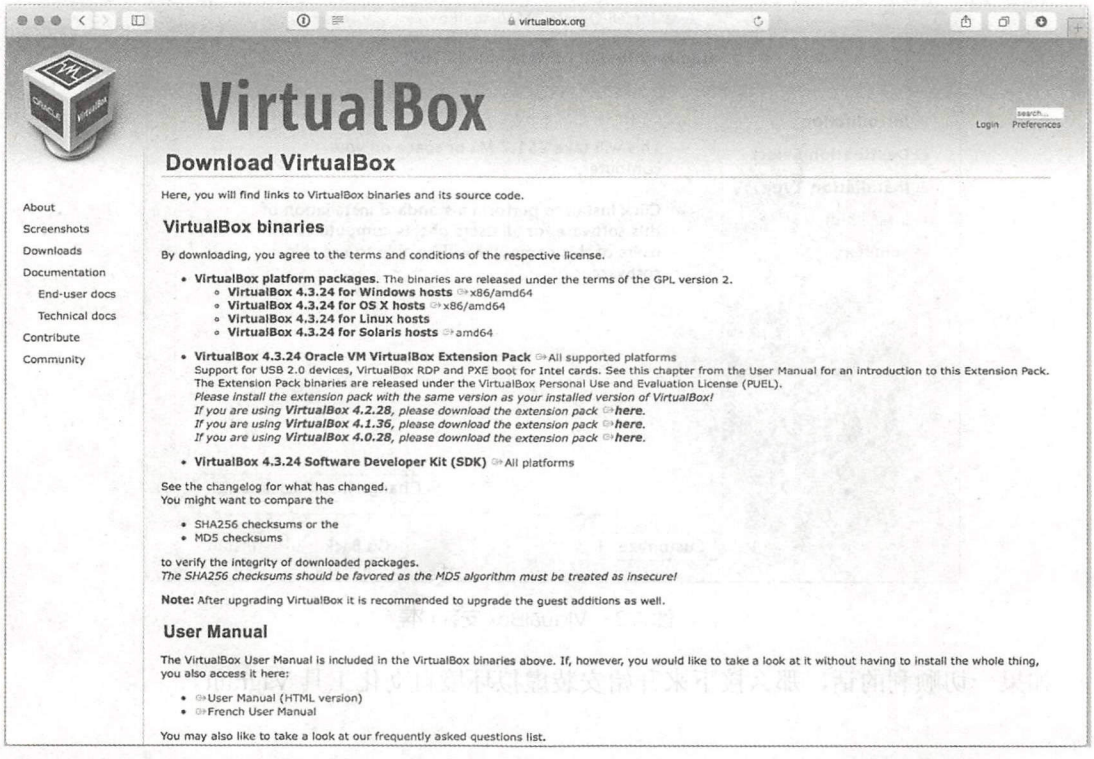


图 A.1 VirtualBox 下载页，包含适用于 Windows、OS X、Linux 和 Solaris 的下载链接

注意 在撰写本书时，Vagrant 支持 VirtualBox 的 4.0.x、4.1.x、4.2.x、4.3.x, 5.0.x 和 5.1.x 版本。由于 VirtualBox 和 Vagrant 如此流行，我们相信 Vagrant 将会继续在 VirtualBox 项目新版本发布后不久便予以支持。换句话说，你应该能够下载 VirtualBox 最新版本，而不必担心 Vagrant 的不兼容性。

在下载软件包后，请运行安装程序。尽管每个操作系统都有些许不同，但安装过程应该是一样的。在运行安装向导时，大多数情况下遵循默认安装选项即可（见图 A.2）。

如果你想更深入地了解用于安装 VirtualBox 的可用选项，VirtualBox 用户手册的第 2 章有非常详细的安装介绍。具体内容详见 www.virtualbox.org/manual/ch02.html。

如果你在安装 VirtualBox 时遇到了问题，最好去 VirtualBox 社区请求帮助。如果你遇到任何问题，www.Freenode.net 上的邮件列表、论坛和 #vbox IRC 频道都是很好的资源，可助你一臂之力。

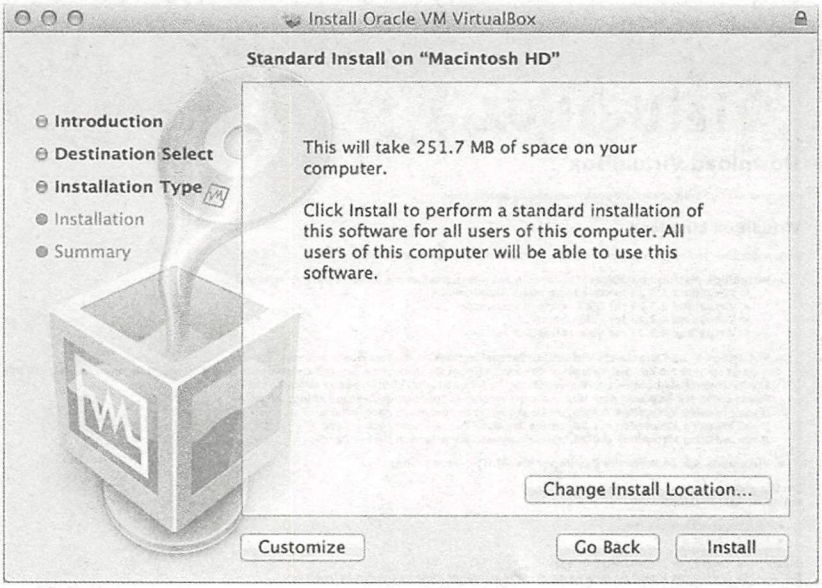


图 A.2　VirtualBox 安装向导

如果一切顺利的话，那么接下来开始安装虚拟环境自动化工具 Vagrant。

A.2　安装 Vagrant

Vagrant 是一款用于管理虚拟环境的自动化工具。它可以从头开始配置虚拟机，提供用于下载和安装基础虚拟机镜像的结构，还能与 Chef 和 Puppet 等配置管理工具进行集成。这样做带来的结果就是虚拟机部署的一致性，能为开发提供稳定环境。此外，由于运行在本地机器上，因此它会将对 VM 中运行服务的网络访问映射到本地计算机的 `localhost` 网络接口。这一功能使得 RabbitMQ 和其他本书用到的网络工具就好像运行在你的计算机上，而非虚拟机中。

请在浏览器中访问 www.vagrantup.com（见图 A.3），下载适合你电脑的 Vagrant 版本。

点击下载按钮时，你将会看到包含各种版本的下载列表。单击列表顶部的最新版本，你将看到一组特定操作系统安装包（见图 4）。选择适合你计算机的版本，并进行下载。

注意　虽然 VirtualBox 支持 Solaris 作为主机操作系统，但是 Vagrant 只支持 Windows、OS X 和少数 Linux 发行版。

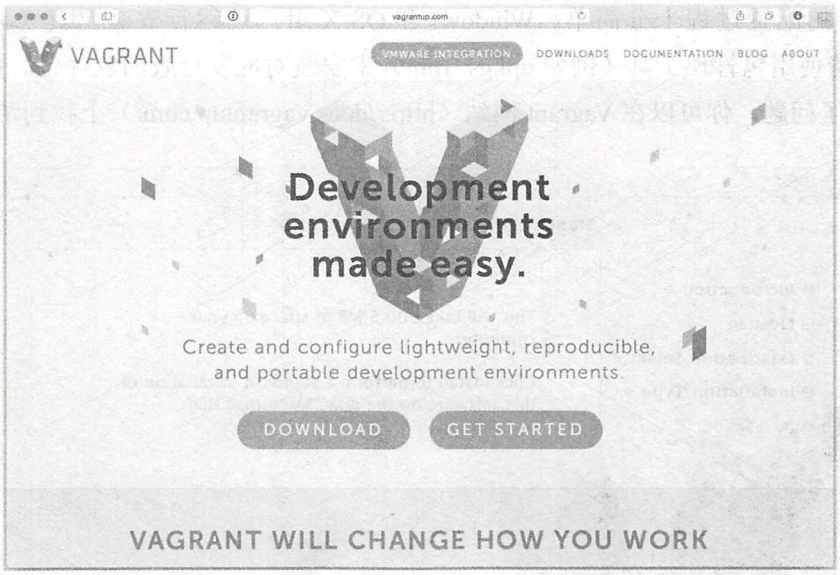


图 A.3 VagrantUp.com 上 Vagrant 项目的主页

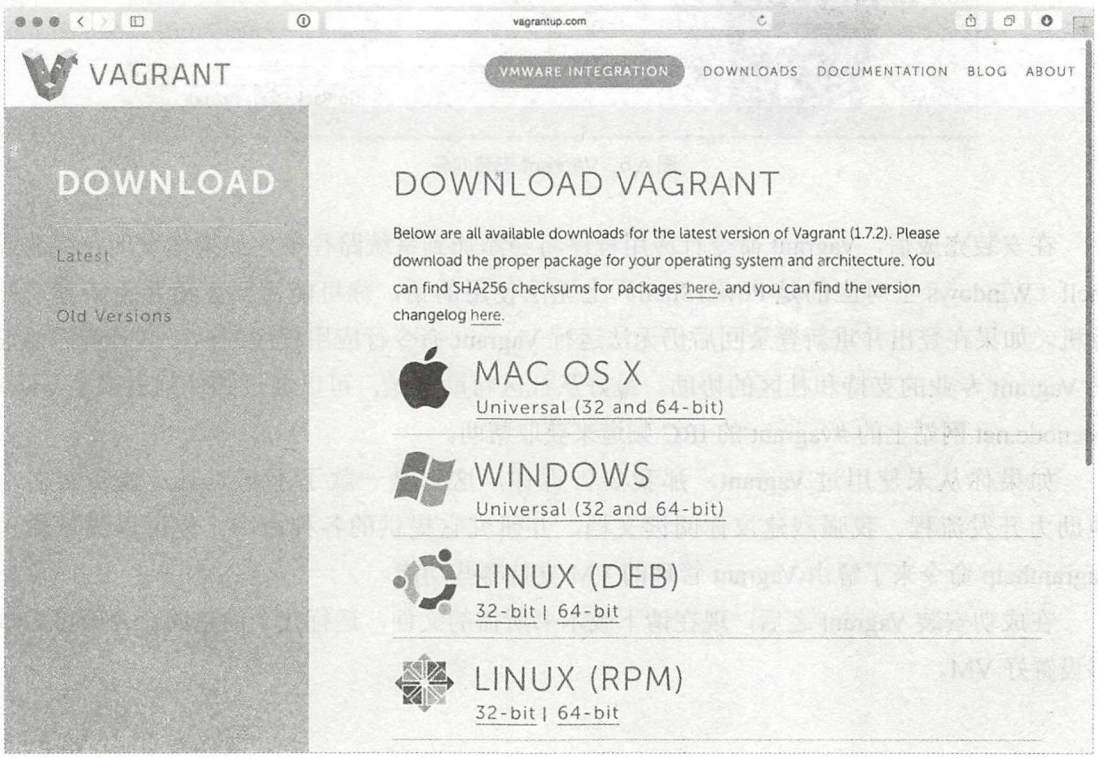


图 A.4 Vagrant 下载页

Vagrant 的配置过程十分简单。Windows 和 OS X 用户请运行安装工具（见图 A.5）。Linux 用户请使用包管理工具（例如 dpkg、rpm）来安装对应发行版的软件包。如果在安装过程中遇到了问题，你可以在 Vagrant 网站（http://docs.vagrantup.com/）上找到帮助文档。

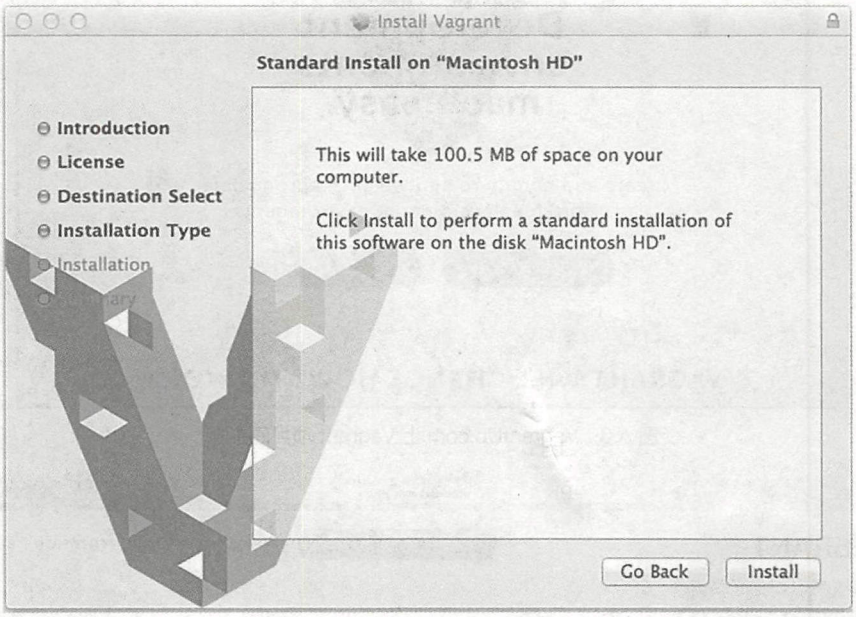


图 A.5　Vagrant 安装向导

在安装完成后，Vagrant 命令行应用程序将会添加到系统路径中。如果你发现在终端或 shell（Windows 上对应的是 PowerShell）上无法使用的话，你可能需要注销并重新登录计算机。如果在登出并重新登录回后仍无法运行 Vagrant 命令行应用程序的话，这时就需要求助 Vagrant 专业的支持和社区的协助。最好从社区帮助开始，可以通过邮件列表或是 www.freenode.net 网站上的 #vagrant 的 IRC 频道来获取帮助。

如果你从未使用过 Vagrant，那我要告诉你，这真是一款了不起工具，能实实在在地助力开发流程。我强烈建议你阅读文档，并研究它提供的各种命令。你可以尝试输入 vagranthelp 命令来了解由 Vagrant 管理的 VM 支持哪些功能。

在成功安装 Vagrant 之后，现在请下载本书所需的文件，运行几个简单的命令即可为本书设置好 VM。

A.3 设置 Vagrant 虚拟机

下一步，你需要下载一个小型 zip 文件，其中包含了本书所使用的虚拟环境所需的 Vagrant 配置和支持文件。在第二篇中的群集教程中定义了多个 VM。除非有特殊说明，否则请使用主 VM。

首先，请从代码文件中下载 Vagrant 配置文件、rmqid-vagrant.zip。我们将用这些文件来配置环境。

在下载了压缩文件之后，请将文件的内容解压缩到一个目录。你需要记住这个目录，并能通过终端或是 PowerShell（如果你是一位 Windows 用户）进行访问。zip 文件解压缩后的内容位于 rmqid-vagrant 的目录下。请打开终端，并切换到该目录下。请输入下列命令来启动 Vagrant 配置的主 VM：

```
vagrant up primary
```

这个过程平均需要 10 到 15 分钟，并依赖计算机的处理速度和网络连接的不同而不同。当第一次启动该进程时，你应当能在控制台上看到表示 VM 启动进度的输出内容，如下所示：

```
Bringing machine 'primary' upwith 'virtualbox' provider...
==> primary: Box 'gmr/rmqid-primary' could not be found. Attempting to find
     and install...
    primary: Box Provider: virtualbox primary: Box Version: >= 0
==> primary: Loading metadatafor box 'gmr/rmqid-primary'
    primary: URL:https://atlas.hashicorp.com/gmr/rmqid-primary
==> primary: Forwarding ports... primary: 1883 => 1883 (adapter1)
    primary: 22 => 2222 (adapter 1)
==> primary: Booting VM...
==> primary: Waiting for machine toboot. Thismay take a few minutes...
    primary: SSH address: 127.0.0.1:2222
    primary: SSH username: vagrant primary: SSH auth method: private key
==> primary: Running provisioner:shell... primary: Running:inline script
==> primary: stdin: is not atty
==> primary:From https://github.com/gmr/RabbitMQ-in-Depth
==> primary:* branch      master  -> FETCH_HEAD
==> primary:80e7615..469fc8cmaster       -> origin/master
==> primary: Updating 80e7615..469fc8c
==> primary: Fast-forward
```

如果你看到的输出结果和上述内容不同，那么在你的计算机上可能已经绑定并监听了 VM 正在尝试使用的其中一个端口。如果你在本地机器上已经运行的 RabbitMQ 的话，那么请关闭它，然后尝试再次运行 vagrant。VM 将尝试使用端口 188、2222、5671、5672、8883、8888、9001、15670、15671、15672 和 61613。端口真的不少。不过，你正在本机上

运行一个虚拟服务器，其中包含若干个服务，用于本书中的示例。为了使 VM 正常工作，你需要停止在这些端口上监听的所有应用程序。

此外，如果你在 Windows 计算机上进行安装的话，那么防火墙可能会提示你是否允许与 VM 建立链接。请确保你允许上述链接，否则你将无法连接到虚拟机，而 Vagrant 也无法配置它。

如果一切正常，机器也启动成功，在 VM 出现停顿或没有继续下去时，这时可能会有一些配置需要进行设置。一旦 VM 设置完成，你将回到控制台中的提示符处。

最后，如果你想要停止 VM 的话，请输入命令 vagrant halt 即可。

现在，让我们在浏览器里输入一些 URL 来确认我们的配置是否一切正常。

A.4 确认安装

我们需要确保以下两个应用程序的设置是否正确。只有如此，一切才能按预期的方式工作，你才能着手运行书中的示例程序。

首先是 RabbitMQ。请打开浏览器并访问 http://localhost:15672，以便检查 RabbitMQ 是否正确安装。你应当能看到如图 A.6 所示的界面。

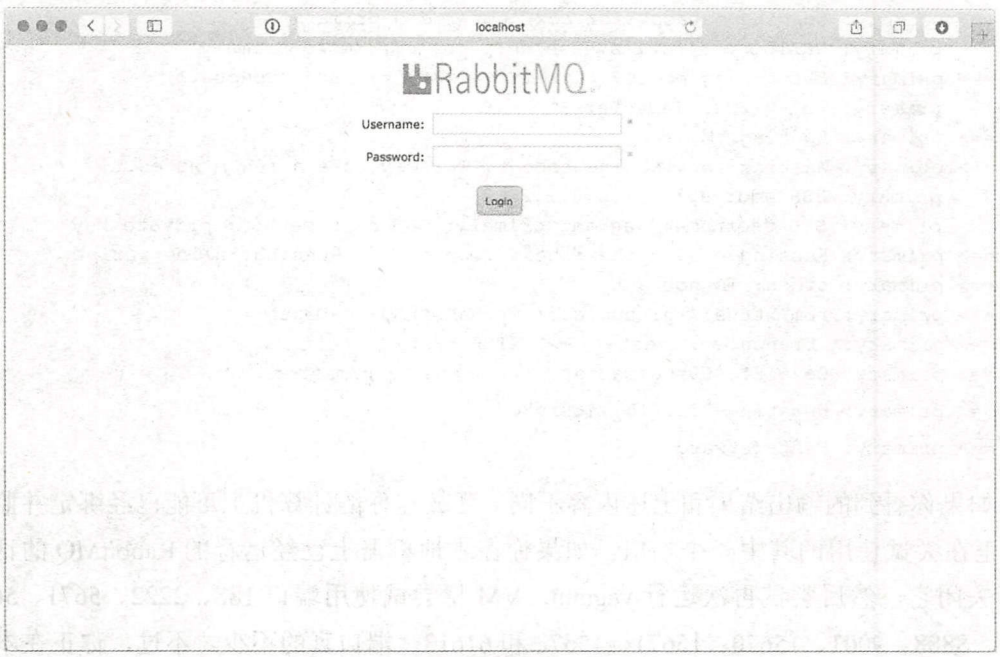


图 A.6 RabbitMQ 管理界面登录窗口

登录管理界面的默认用户名和密码是“guest”和“guest”。登录成功之后，你将能看到管理界面的主页，它将呈现服务器配置和状态的概览。

在确认了 RabbitMQ 之后，另一个需要检测的应用是 IPython Notebook 服务器。该应用程序允许你在 Web 浏览器中以交互方式运行基于 Python 的代码示例。在该虚拟机中，所有代码清单和示例均在 IPython Notebook 服务器中。这样做使得你能以相互独立的方式来打开并运行程序。IPython Notebook 服务器监听的端口是 8888。请在浏览器中打开新标签页，访问 http://localhost:8888。你看到的页面类似于图 A.7。

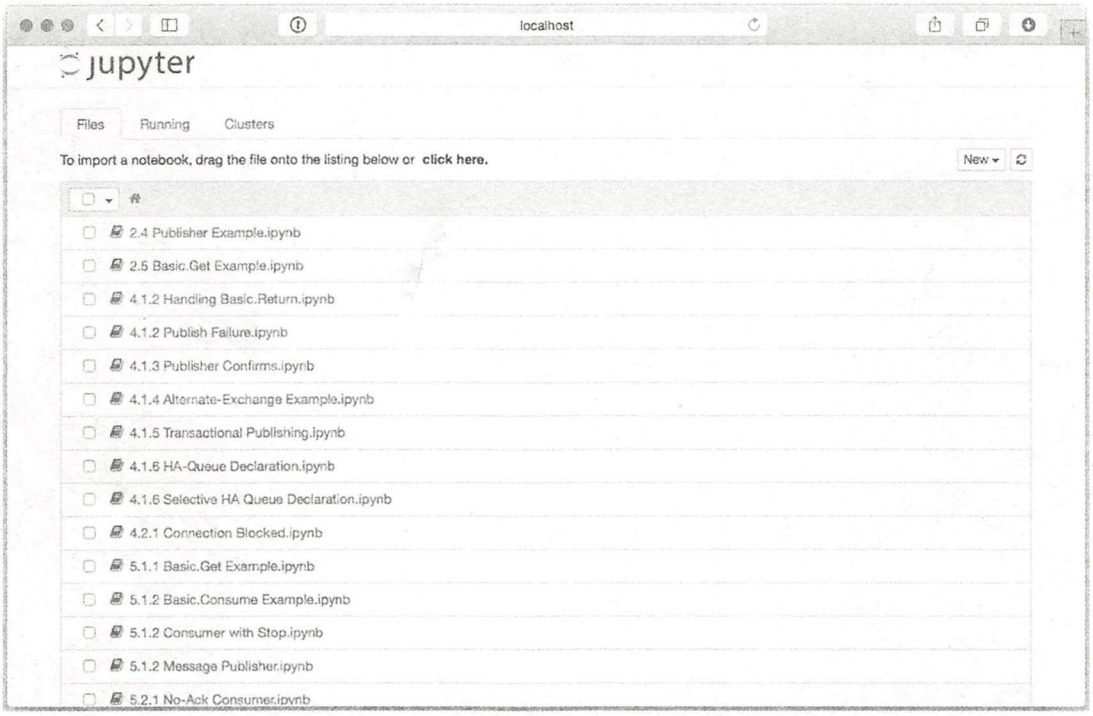


图 A.7 IPython Notebook 服务器首页上展示了本书所有的代码文件

如果你对 IPython Notebook 服务器的功能或文档感兴趣的话，它的网站上包含了丰富的信息(http://ipython.org)。这是一款非常有用的应用程序，在科学和数据处理社区中广受欢迎。人们用它来处理数据集和执行数据可视化。

A.5 小结

现在，你应当已经成功运行了本书的 VM。让我们再回顾一下。你安装的 VirtualBox 是一款虚拟环境自动化工具。你下载了本书 Vagrant 配置和 Chef 参考手册，并能通过使用 `vagrant up` 命令来启动新的 VM。这一切就绪之后，你就能使用 VM 里的工具来跟随本书的脚步，开启你的 RabbitMQ 之旅了。

Broadview® www.broadview.com.cn 博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

深入RabbitMQ

RabbitMQ In Depth

大部分现代分布式应用的核心就是队列。它提供了缓存、优先级区分和消息路由的能力。RabbitMQ是一款高性能的消息代理服务器，基于高级消息队列协议。它经受住了实践的检验，足够快速，足够强大，几乎可以满足所有消息投递的需要。仅需要一些简单的设置，就能立即使用它管理低级别服务通信、应用集成和分布式系统的消息路由。

本书是构建和维护基于消息的应用程序的实用指南。本书详细介绍了RabbitMQ，其中重点介绍了它的工作机制。不论是简单的网络服务，还是复杂的分布式设计，都可以从中找到真实系统的示例与详细解释。还可以从中领略到核心架构决策和有效运营管理流程开发所需的深刻见解。

本书包括：

◎AMQP协议（Advanced Message Queueing Protocol）
◎使用MQTT、Stomp和HTTP进行通信
◎非常有价值的故障诊断技术
◎数据库集成

本书适合那些对面向消息的系统有一定基础的开发者阅读。

Gavin M. Roy是一位积极主动的开源传播者和倡导者，自20世纪90年代中期以来一直从事互联网和企业技术方面的工作。技术编辑James Titcumb是一名自由开发者、培训师、演讲者，并且是开源项目的积极贡献者。

“对初学者和专家来说都是很好的资源……本书展示了如何将RabbitMQ集成到成功的企业应用程序中。”

——来自惠普公司的Ian Dallas

“RabbitMQ全面详尽的参考手册。从术语到代码，再到模式，应有尽有！”

——来自Quantum Metric的Andrew Meredith

“用于起步及排查迁往RabbitMQ过程中故障的速查手册。”

——来自巴塞罗那拉萨尔大学的Nadia Noori

“实乃真知灼见。”

—来自Mozzart Bet的Miloš Milivojević

MANNING

策划编辑：张春雨
责任编辑：牛　勇
封面设计：李　玲

上架建议：网站架构 / 开发

ISBN 978-7-121-34180-9

定价：79.00元